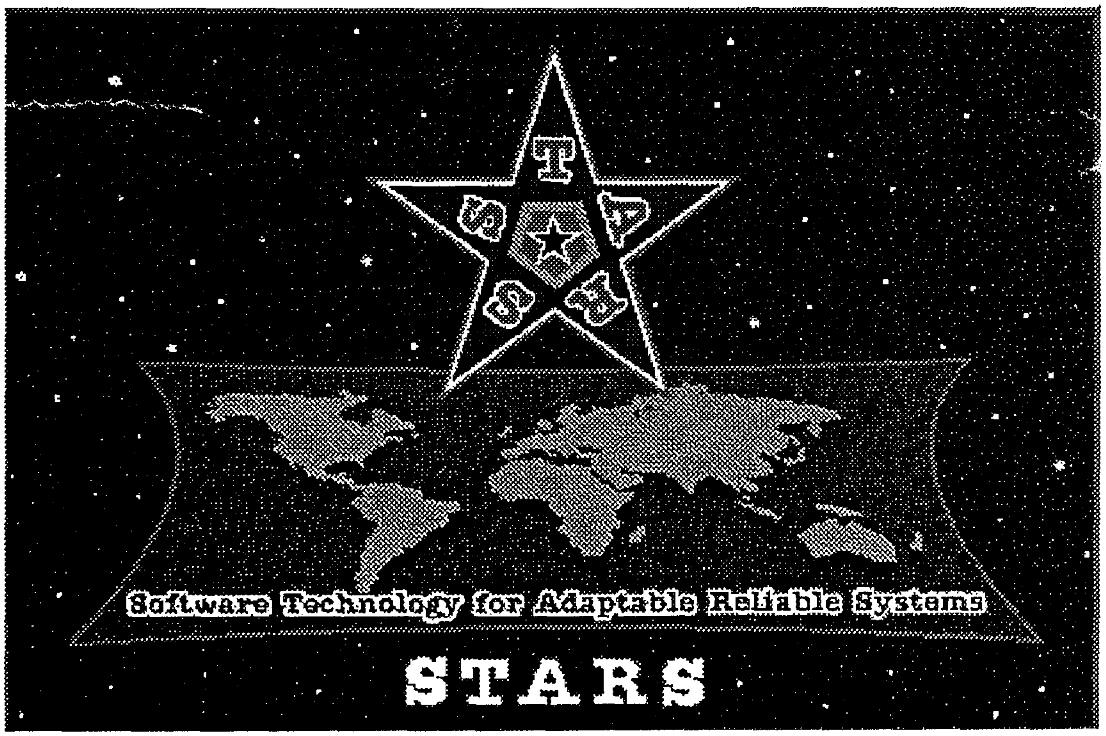
TASK: PA18
CDRL: C001
02 September 1994

Penelope Reference Manual, Version 3-3

(subsumes Data Item C002, LarchAda Specification Manual for Sequential Ada, Chapters 3-6)

Informal Technical Data



STARS-AC-C001/001/00

REPORT DOCUMENTION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE
02 September 1994

3. REPORT TYPE AND DATES COVERED
Informal Technical Report

4. TITLE AND SUBTITLE
Penelope Reference Manual, Version 3-3 (C001), includes Larch/Ada Specification Manual for Sequential Ada (C002)

5. FUNDING NUMBERS
F19628-93-C-0130

6. AUTHOR(S)
Carla Marceau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091-3499

8. PERFORMING ORGANIZATION REPORT NUMBER
CDRL NBR
STARS-AC-C001/001/00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
ESC/ENS
Hanscom AFB, MA 01731-2816

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
C001 & C002

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution “A”

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Penelope is an interactive environment that helps its user to develop and verify programs written in a rich subset of sequential Ada. Penelope is well-suited to developing programs in the goal-directed style advocated by Gries and Diijkstra. In this style the programmer develops a program from a specification in a way that ensures the program will meet the specification. Of course, Penelope can also be used to verify previously written programs. With Penelope, it is often possible to modify a verified program and verify the modified version with minimal effort by replaying and modifying the original program's proof.

14. SUBJECT TERMS

15. NUMBER OF PAGES
140

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: PA18
CDRL: C001
02 September 1994

INFORMAL TECHNICAL REPORT

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Penelope Reference Manual, Version 3-3
(subsumes Data Item C002, Larch/Ada Specification Manual for Sequential Ada, Chapters 3-6)

STARS-AC-C001/001/00
02 September 1994

CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Odyssey Research Associates
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

TASK: PA18
CDRL: C001
02 September 1994

Data Reference: STARS-AC-C001/001/00
INFORMAL TECHNICAL REPORT
Penelope Reference Manual, Version 3-3
(subsumes Data Item C002, Larch/Ada Specification Manual for Sequential Ada, Chapters 3-6)

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address: delivery@stars.reston.paramax.com

TASK: PA18
CDRL: C001
02 September 1994

Data Reference: STARS-AC-C001/001/00
INFORMAL TECHNICAL REPORT
Penelope Reference Manual, Version 3-3
(subsumes Data Item C002, Larch/Ada Specification Manual for Sequential Ada, Chapters 3-6)

Principal Author(s):

Carla Marceau *Date*

Approvals:

Program Manager *Teri F. Payton* *Date*

(Signatures on File)

TASK: PA18
CDRL: C001
02 September 1994

Data Reference: STARS-AC-C001/001/00
INFORMAL TECHNICAL REPORT
Penelope Reference Manual, Version 3-3
(subsumes Data Item C002, Larch/Ada Specification Manual for Sequential Ada, Chapters 3-6)

Abstract

This manual documents the language and commands of the Penelope environment. It is intended for the Penelope user who desires to write specifications, develop programs, and carry out correctness proofs using Penelope. The user is assumed to be an Ada programmer with a strong mathematical background. A working knowledge of predicate calculus (such as provided in [6, Ch. 2]) is essential. The manual is primarily tutorial in nature. It presents the features of Penelope together with some idea of how to use Penelope to develop and verify programs.

TASK: PA18
CDRL: C001
02 September 1994

Data Reference: STARS-AC-C001/001/00
INFORMAL TECHNICAL REPORT
Penelope Reference Manual, Version 3-3
(subsumes Data Item C002, Larch/Ada Specification Manual for Sequential Ada, Chapters 3-6)

Change Record:

Data ID	*Description of Change*	*Date*
STARS-AC-C001/001/00	Describes software upgrade to version 3-3.0	02 September 1994
STARS-AC-A023/004/00	Original Issue	26 February 1994

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Penelope is an interactive environment that helps its user to develop and verify programs written in a rich subset of sequential Ada. Penelope is well-suited to developing programs in the goal-directed style advocated by Gries [6] and Dijkstra [2]. In this style the programmer develops a program from a specification in a way that ensures the program will meet the specification. Of course, Penelope can also be used to verify previously written programs. With Penelope, it is often possible to modify a verified program and verify the modified version with minimal effort by replaying and modifying the original program's proof.

This manual documents the language and commands of the Penelope environment. It is intended for the Penelope user who desires to write specifications, develop programs, and carry out correctness proofs using Penelope. The user is assumed to be an Ada programmer with a strong mathematical background. A working knowledge of predicate calculus (such as provided in [6, Ch. 2]) is essential. The manual is primarily tutorial in nature. It presents the features of Penelope together with some idea of how to use Penelope to develop and verify programs.

Chapter 2 documents how to get Penelope running. In order to develop and verify programs using Penelope, though, you will need to understand Penelope's approach to formal verification and to simplification and theorem proving.

1.1 Formal specification in Penelope

Here is a simple example of a specification in Penelope:

```
procedure prime_flag(x: in integer; b: out boolean);
  --| where
  --|    in 0 <= x and x <= 1000;
  --|    out b = is_prime(x);
  --| end where;
```

This specification says that, on entry, the value of `x` must not be negative or too large, and on normal exit the value of `b` should be the value of *is_prime(x)*. The specification is written in a language called *Larch/Ada*, to which much of this manual is devoted. When we specify a subprogram like `prime_flag`, we write down what must be true on entry to the subprogram and what we want to guarantee on termination. Because the annotation does not explicitly mention the possibility of exceptional exit, it implicitly asserts that execution of `prime_flag` will not propagate an exception.[1] Larch/Ada also permits us to describe the behavior of subprograms that do propagate exceptions.

Specification and verification in Penelope are *formal.* We write specifications in a formal mathematical language and carry out rigorous proofs of correctness. Functions like *is_prime* above are defined axiomatically. The proposed theorems that we have to prove are generated by Penelope based on a denotational definition of the semantics of Ada.[2]

Penelope guarantees only *partial correctness*. That is, if we verify a program using Penelope, we know that *if it terminates*, either normally or by raising an exception, then the specified conditions will hold. We do not, however, guarantee that it will terminate. Later versions of Penelope will provide the capability of proving termination.

A description of the specification language takes up much of this manual. The specification language is described in Chapters 3 through 7. Penelope uses Larch's two-tiered approach to specification. A mathematical tier defines mathematical domains and functions. An interface tier uses these functions to specify what the program should do. The two tiers share a common term language: the mathematical tier defines the meaning of the terms and the interface tier uses them to specify programs. Various chapters of this manual describe the mathematical domains (type system), the common terms, the interface language, and the language for specifying mathematics.

1.2 What Penelope does

Penelope provides an interactive environment for developing Ada programs and specifications. The Penelope user typically works on one compilation unit at a time. Penelope supports both syntax-directed editing and text editing; it is also possible to write a program using a general editing program such as Emacs and read it in. The subset of Ada that Penelope currently supports is described in Appendix B.[3]

Verification in Penelope follows a familiar model. The user specifies, for example, conditions on the state in which a subprogram may be called (*entry conditions*) and what should be true

[1]Verification in Penelope applies to executions during which neither storage error nor numeric overflow occurs. See Appendix B for other restrictions on the current version of Penelope.

[2]The denotational definition covers most of sequential Ada. Work is still in progress on concurrent Ada.

[3]Penelope type-checks the program and the specification, but does not currently support the full static-semantic checking of Ada.

when it terminates either normally (*exit conditions*) or by raising an exception. Penelope computes an (approximately) weakest *precondition* of the program with respect to the exit conditions. A weakest precondition is a condition that must be true on program entry in order for the exit conditions to be guaranteed to hold on exit.[4]

Penelope computes the precondition of a program by computing the precondition of each statement and declaration. Users acquainted with the work of Floyd [3], Hoare [13], Dijkstra, or Gries will recognize the computed preconditions, which intuitively represent the assertions that must hold at each control point, based on the semantics of the language.[5] Penelope computes preconditions incrementally, which means that every time a programmer makes a change to a program, the preconditions immediately reflect the effects of that change. The user can inspect these computed preconditions and can use them in developing the program, in the style of Gries [6].

Using the computed preconditions, Penelope generates *verification conditions*, usually one per loop plus one per subprogram body. The verification conditions are purely logical statements (boolean terms) that, if true, guarantee that the program satisfies its specification. The verification condition for a subprogram body, for example, states that the entry conditions in the subprogram annotation are sufficient to prove the computed precondition of the subprogram. That is, we know from the specification what should be true when the program terminates (exit condition); Penelope computes what *must* be true at the beginning of the program for that exit condition to hold (the precondition); and we must show (by proving the verification condition) that the entry conditions are sufficient to guarantee the precondition.

1.3 Simplification and proof

We would like Penelope to automatically *simplify* the preconditions that it computes, putting them in the most convenient form, and to automatically prove the verification conditions if possible. Unfortunately, all but the most trivial simplification and proof in Penelope require the guidance and control of the user. This interaction is necessary because of the well-known fact that simplification and theorem proving are in general undecidable; even so-called automatic theorem provers usually require a good deal of guidance from human beings.

Penelope includes a simple proof editor/checker for arithmetic and predicate calculus, which provides a number of proof rules for simplification and proof, described in this manual. Penelope applies the simplification and proof rules according to user directions (there is a menu of proof rules) and shows the user what, if anything, still has to be proved after each step. This internal proof editor is discussed in Chapter 8.

Proof of the verification conditions will appeal to an underlying body of mathematics—for

[4]The computed preconditions correspond to Dijkstra's function *wlp* [2].

[5]The defining semantics for Ada constructs used by Penelope is documented in denotational semantic style in [18].

example, the definition of *is_prime* and facts about prime numbers. This mathematics may be assumed or may be developed with Penelope's theorem prover.

In summary, Penelope is the user's trained assistant in verification. It performs well-defined but tedious tasks (like computing verification conditions and carrying out proof steps) while the user is responsible for the intelligent part of the work: specifying the program, developing the program, and deciding how to prove it.

1.4 Organization of this manual

This manual has eight chapters and three appendices:

- Chapter 1 is an introduction to Penelope's approach to formal specification and to simplification and theorem proving.
- Chapter 2 describes the Penelope buffer; starting Penelope, saving your work, and exiting Penelope; the basics of the user interface; the Penelope library; and checking the verification status of a program.
- Chapter 3 covers lexical matters.
- Chapter 4 informally presents Penelope's approach to specification and introduces the type system of the specification language.
- Chapter 5 describes the mathematical *terms* used in specifying programs.
- Chapter 6 describes Larch/Ada, a Larch interface language for specifying Ada programs.
- Chapter 7 describes the Penelope variant of the Larch Shared Language, a language used for defining the mathematics involved in a specification.
- Chapter 8 describes Penelope's internal proof editor.
- Appendix A is a sample verification using Penelope.
- Appendix B shows the subset of Ada that Penelope currently supports.
- Appendix C is a summary of proof rules.

1.5 Other sources of information

Earlier versions of the material contained in this document were combined with motivational material and some description of the semantics of Larch/Ada in *A Short Introduction to Larch/Ada-88*. That material is now found in [17]. Overviews of Penelope can be found in [9] and [16]. Documentation of the mathematical foundations of Penelope can be found in

[7, 8, 9, 12, 18]. A description of and manual for Penelope's user interface can be found in the Synthesizer Generator manual [5]. (Portions of that manual may be found in Penelope's on-line help facility.)

1.6 Syntax notation

The context-free syntax used in this manual is a simple variant of Backus-Naur form. In particular,

- Lowercase italic words in angle brackets are used to denote syntactic categories, for example, $\langle \textit{real literal} \rangle$.
- Boldface type denotes reserved words, for example **invariant**, or other keywords.
- A vertical bar separates alternative items.
- The following special symbols are used:

$[[\alpha]]$	optional occurrence of α
$[[\alpha]]^*$	zero or more occurrences of α
$[[\alpha]]^+$	one or more occurrences of α
$[[\alpha]]^*_\beta$	$[[\alpha[[\beta\alpha]]^*]]$ (β is a separator)
$[[\alpha]]^+_\beta$	$\alpha[[\beta\alpha]]^*$

For example, the first of the following rules states that a (function application) term consists of a function designator followed by a list of zero or more terms enclosed in parentheses. The second states that a $\langle \textit{varlist} \rangle$ (list of logical variables) consists of one or more identifiers, with optional sortmarks. The third rule states that a real literal may include an optional exponent.

$$\begin{aligned}
\langle \textit{term} \rangle &::= \langle \textit{designator} \rangle \ (\ [[\langle \textit{term} \rangle]]^*_, \ ) \\
\langle \textit{varlist} \rangle &::= [[\langle \textit{identifier} \rangle [[:\langle \textit{sortmark} \rangle]]\ ]]^+_, \\
\langle \textit{real literal} \rangle &::= \langle \textit{integer} \rangle \,.\, \langle \textit{integer} \rangle\ [[\ \langle \textit{exponent} \rangle]]
\end{aligned}$$

1.7 Font conventions

In Penelope, the Ada programming language is intermixed with specifications and proofs, written in mathematical language. Following Penelope's default, the examples in this manual use the following conventions:

- Ada code is in typewriter font (e.g., `z:= x + y;`).

- Specifications are in italics (e.g., $z = x + y$).
- Proofs are in sans serif font.

The font has no semantic significance; the meaning of an operator symbol is determined by context.

Chapter 2

Using Penelope for verification

Verifying a program with Penelope involves two kinds of activities: we have to verify individual compilation units and the initial elaboration of the main program; and, as in compilation, we want to keep the results of our previous work in a library to be used in further work.

The first kind of activity takes place in the context of the Penelope *buffer*. In this chapter we first describe how to get Penelope started. We then describe the Penelope user interface. Next we introduce the structure of the Penelope buffer, including some practical advice on using some of the constructs mentioned. Then we show how to save your work and exit Penelope. Two final sections discuss the organization of the library and the bookkeeping necessary in reverification.

2.1 Starting Penelope

Penelope runs under Unix and X Windows 11R5. As a practical matter, we provide some brief instructions on how to set up an environment so you can run Penelope.

Penelope was created using Version 4.1 of the Synthesizer Generator. To get Penelope running, the environment must be set up appropriately.

1. Edit the release file called `syngen_resources`, changing the pathname `/usr/local/src/syn/helpdocs/SystemDoc` to the location where the SystemDoc directory of the Penelope release is located in your installation.

2. Put a copy of the release file `syngen_resources` in some convenient directory. Your home directory is a good choice, allowing later individual customization of Penelope styles.

3. Execute a command to make Penelope resources known to X:

   ```
   xrdb -merge syngen_resources
   ```

You may need to use a full path name for syngen_resources. You will probably want to put such a command in your window initialization file. Otherwise, you may need to execute the xrdb command before each invocation of Penelope.

The resources in syngen_resources initialize the Penelope window to a rather large size and tell Penelope where to find the on-line help for Synthesizer Generator commands. You can edit this file to use different font size or colors, use a smaller Penelope window, and so on. Other resources that can be set in this way are documented in [5, Ch. 4].

To run Penelope, two files must be on your search path: penelope itself and the simplifier distributed with Penelope. Two versions of the simplifier are distributed with Penelope: one is better for programs that include real arithmetic; the other is preferable for other programs.

You invoke Penelope with or without a single filename argument. If you invoke Penelope without an argument, you can use the open command to read in a file once Penelope is running. You can set the right margin and other parameters once Penelope is running by using the set-parameters command on the Options menu. Default values are provided in the syngen_resources file.

It is not a good idea for new Penelope users to develop a program using a text editor and then try to read it into Penelope. First of all, it is usually easier to develop a verified program—that is, to develop the specification, the program and the proof together—than to verify a previously written program. Also, Penelope expects to read a program with specification and proof; parsing problems may arise in trying to read unannotated programs. Experienced Penelope users are more familiar with Penelope's expectations and can more easily alter the program being read in to make it acceptable.

As an aid to reading information in the Penelope buffer, Penelope uses different styles and colors to present information. If you follow the installation directions and Penelope comes up with boldface keywords and text in color (if you have a color monitor) or italics, then you may ignore the remainder of this paragraph. If you invoke Penelope and the buffer has no bold style and no color or italics, make sure that you have made Penelope resources known to X through the xrdb command. If that is not the problem, consult your system administrator. It is worth the effort to get the different styles to work. With program, specification, and proof interspersed in the same buffer, the styles make it much easier to understand the buffer contents.

You may change the fonts or colors for your version of Penelope. The syngen_resources file associates style names with X font information. The following are the styles in the syngen_resources file and their use:

Normal	Ada program text
Keyword	Ada keywords
Spec	Larch/Ada specification
SpecKey	Keywords of Larch/Ada
Proof	Proofs

After you have Penelope running and set up with your style choices, you may want to read in the factorial example provided with the Penelope release to follow the example later in this chapter. First, you will need to set up a directory to work in, with a subdirectory named `lib`, containing the file `Factorial.trait.lib` from the Penelope release. The file `factorial-example` from the release should be in your working directory. You can then start Penelope with the command

```
penelope factorial-example
```

Or you can start Penelope without the filename argument, and instead use the `open` command on the `File` menu. The `open` command creates a window where you give the pathname for the `factorial-example` file and click on the `start` button.

2.2 Penelope's user interface

The details of Penelope's user interface are described in [5], which defines how commands are invoked, how programs are edited, how the cursor is moved from one point to another, and so on. These details of the user interface have nothing to do with verification but are essential for actually using Penelope. It will help if you are familiar with the Emacs editor because Penelope's text editing commands are similar to those in Emacs. Penelope includes an on-line help facility for editing commands. The `Help` button or the `describe-command` command on the `Help` menu invoke this facility, which provides descriptions and some tutorial overview for many navigational and editing commands. No on-line help is yet available for the Penelope commands. The `help-for-editor`, `tutorial`, and Penelope-specific `describe-` commands on the `Help` menu are not yet implemented.

Penelope supports both syntax editing and text editing. The left mouse button is used to select for editing. The syntactic selection is indicated by underlining. The current text selection is in reverse video. Within a highlighted area you can use the Synthesizer Generator's text editing capabilities to modify the text, which is reparsed when you hit the `return` key. A caret marks the current insertion point, where text will be inserted.

2.2.1 The help-pane menu of transformations

At the bottom of the screen a *help-pane* tells at which non-terminal (`context`) the cursor is positioned. The help-pane presents editing commands called *transformations*. The transformations presented in the help-pane depend on the current selection. If you need more space

to view all the transformations, you can drag on the black square at the right of the line dividing the window and the help-pane. You click on a help-pane item with the left mouse button to select a transformation. Transformations fall into certain categories:

template A *placeholder* (e.g. `<statement>`) indicates a non-terminal that has yet to be expanded. When the cursor is positioned at a placeholder, transformations appear in the help-pane corresponding to the possible expansions of the current non-terminal. The names of such help-pane items should be self-explanatory. For example, when the cursor is positioned at a statement placeholder, the transformations `while-loop` and `if-then-else` appear, as well as others. Each such transformation causes a *template* for the desired construct to replace the placeholder. You can then fill in the required information (e.g., the expression and statement sequence for a while loop).

optional items Optional non-terminals do not always appear in templates. For example, ⟨*later declarative item*⟩s are not present in the template for subprograms. To obtain a template for a later declarative item, you click on the subprogram, or at the place where a later declarative item might occur. The help-pane then shows a transformation for later declarative items. The names of such help-pane items should be self-explanatory. Sometimes more than one help-pane item appears for an optional item; in this case you can click on any of the help-pane items.

lists When the cursor is positioned at an item of a list, transformations `insert-before` and `insert-after` appear in the help-pane, enabling you to create placeholders for new items in the list. You can also click between list items to get a menu-item for a list item.

proof-rules Transformations are the way in which we tell Penelope to initiate simplification and proofs and how to carry them out. These transformations are described in Chapter 8.

replacements A few transformations replace the current selection with something you might want instead. These transformations are documented in relevant sections of this manual.

2.2.2 Command menu

Most of the commands on the pull-down menus are defined by the Synthesizer Generator (see [5, Ch. 3]). The `Penelope` menu, however, contains commands peculiar to Penelope, which are described in this manual:

`about-penelope`	Displays Penelope license restrictions
`write-library`	Updates library—see Section 2.5
`write-hol`	Writes theory in HOL format—not currently available

2.2.3 Views on the buffer

There is a lot of information in the Penelope buffer, and it can sometimes be useful to look at a subset of the information there. Three alternative *views* of the buffer are available: the AdaView, the IncompleteProofs view, and the Internal View. The default view, shown in Figure 2.2, is called the *BASEVIEW.* To get an alternative view of the buffer, invoke the `change-view` command on the `Window` command menu.

The *AdaView* shows just the Ada program. This is useful when, for example, you wish to compile the program after verifying it. You can switch to the AdaView and then write it out in text format. (Although Penelope annotations and proof lines are included in the buffer as pseudo-comments, successive lines of an annotation or proof step do not always include the comment marker and do not parse correctly.)

The *IncompleteProofs view* shows just proofs that are not yet completed. This view is useful if the buffer is long and the verification status indicates that one or more verification conditions are not proved. If you switch to the IncompleteProofs view, click on the incomplete proof, and switch back, the cursor will be positioned at the offending proof. The normal view is *BASEVIEW.*

The *InternalView* is primarily for use by Penelope developers.

2.3 The Penelope buffer

A Penelope verification takes place in a Penelope *buffer* that contains zero or more Ada compilation units, as well as what looks to an Ada compiler like a lot of Ada comments. To Penelope, these comments include the specification of the program and its proof. Interspersed with Ada compilation units may be "compilation units" that are purely mathematical in nature, providing definitions of terms used in the specification. Indeed, the buffer may contain no Ada code but only only mathematics.

If a Penelope buffer is written out to a text file and then read into a buffer again, Penelope recomputes the verification conditions and rechecks the proof. Thus the Penelope buffer contains all the information required to verify the program fragments it contains. We say "program fragments" because a single buffer typically contains only one or a very few compilation units.

Figure 2.1 shows an example of a Penelope buffer, verifying a factorial function. We will examine this example to see how the buffer is organized and to introduce some of the basic concepts in Penelope. You may wish to study this section after getting the factorial example provided with the Penelope release running (see Section 2.1 on how to get Penelope started).

In the Penelope buffer a small amount of Ada code is surrounded by pseudo-comments. Penelope includes Ada code, specification, proof statements, and error messages. To help distinguish these different kinds of information, Penelope displays them in different *styles*,

```
--| library lib ;
--! Verification status: Not verified
--! Verification status: 2 VCs; 0 VCs hidden; 1 VCs proved
--| with trait Factorial ;
function fact(n : in integer) return integer
  --| where
  --|   in (n>=0);
  --|   return factorial(n);
  --| end where;
  --! VC Status: proved
  --! BY synthesis of TRUE
is
  ntemp : integer := 0;
  ftemp : integer := 1;
begin
  --! VC Status: ** not proved**
  --! BY instantiation of fplus1 in trait Factorial establishing
    --!   1. ftemp=factorial(ntemp)
    --!   2. 0<=ntemp
    --!   >>ntemp>=0
    --! <proof>
  --! THEN
  --! rewriting left to right
  --! BY simplification
  --! BY synthesis of TRUE
  while (ntemp/=n) loop
    --| invariant ftemp=factorial(ntemp) and 0<=ntemp;
    ntemp:=(ntemp+1);
    ftemp:=(ntemp) * ftemp;
  end loop;
  return ftemp;
end fact;
```

Figure 2.1: Example of a Penelope buffer

```
--| Larch
Factorial: trait
  introduces
    factorial: Int -> Int;
  asserts
    forall m:Int
      f0 (rewrite): (factorial(0)=1);
      fplus1: m>=0->factorial(m+1)=(m+1) * factorial(m);
  implies
    forall [m:Int, f:Int]
      fminus1: m>0- >factorial(m)=m * factorial(m-1);
--| end Larch
```

Figure 2.2: A trait for factorial

that is, different colors and or fonts. In this manual examples are displayed using different fonts to distinguish three different areas of the buffer:

Ada The program is in `typewriter` font.

specification The specification is shown here in *italics*. Lines of specification are preceded by --|. When Penelope runs with a color display these items are by default displayed in blue.

proof Elements of the proof of the program are shown here in a sans serif font. On a color display proofs are by default displayed in red. Proof lines begin with `--!` .

The first line in Figure 2.1 tells which library is being used for this verification. The library contains mathematics and information about previously verified compilation units.

2.3.1 Mathematics for factorial

In the verification we rely on the definition of factorial, which is shown in Figure 2.2. This is just the familiar recursive definition for factorial. Penelope follows the Larch style, in which mathematics is developed in units called *traits*. The *Factorial* trait was itself developed using Penelope. Traits and Ada compilation units may be interleaved. (Chapter 7 gives more information about how we write mathematics for Penelope.)

```
function fact(n : in integer) return integer
is
  ntemp : integer := 0;
  ftemp : integer := 1;
begin
  while (ntemp/=n) loop
    ntemp:=(ntemp+1);
    ftemp:=(ntemp * ftemp);
  end loop;
  return ftemp;
end fact;
```

Figure 2.3: Penelope buffer–Ada code only

2.3.2 Verification status of the buffer

The next two lines are Penelope's report on the status of the verification in this buffer. The status can be one of the following:

Verified All verification conditions in this buffer have been proved.

Not Verified If not all verification conditions have been proved, Penelope reports on the total number of verification conditions, how many are currently hidden or ignored, and how many have been proved.

Nothing to verify There are no verification conditions.

The following status line may be present when you are developing mathematics:

Lemma status Penelope warns you if a trait has unproved obligations (e.g., a lemma not proved) or unfinished proofs of lemmas.

2.3.3 Annotations and proof within a compilation unit

The buffer of Figure 2.1 contains a single compilation unit, the function `fact`.[1] It is preceded by an annotation indicating that the trait *Factorial* is used in its specification. The function itself, shown in Figure 2.3, is quite short.

In Ada's syntax the specification of a subprogram declares its name and parameter-result profile. The Ada specification is followed in Penelope by the formal specification, which

[1] In the actual Penelope buffer, expressions are fully parenthesized. In a future version more natural parenthesization will be used. In the examples of this chapter some parentheses are removed for legibility. Note that you do not have to fully parenthesize your input to Penelope; rather Penelope fully parenthesizes its pretty-printing.

is written in Larch/Ada (see Chapter 6). In this manual we will generally use the word *specification* to refer to the formal, Larch/Ada specification.

After the specification of `fact` in Figure 2.1 comes the verification condition for the subprogram. The verification condition consists of a status line followed by a proof. All of these lines begin with the marker `--!`, indicating that they are part of the proof of the program. The proof of this verification condition consists of a single proof step: BY synthesis of TRUE. Since the statement to be proved was simply the term **true**, the verification condition was trivially true. Proofs in Penelope often end with this proof step.

The verification condition status may be one of three things:

proved The verification condition is accompanied by a completed proof.

not proved The accompanying proof is incomplete.

hidden A box ([]) is displayed in place of the proof. Hiding a verification condition can make the Penelope buffer more readable by suppressing sometimes lengthy information.

It is important to remember, however, that if the buffer is written out to a text file while the verification condition is hidden, no record of the proof is contained in the file. A verification condition can be hidden by clicking on the help-pane item `hide-vc` when the cursor is on the verification condition. A hidden verification condition can be redisplayed by positioning the cursor on it and clicking on `show-vc` on the help-pane. (See Section 2.2.1 for further information about the help-pane.)

A second verification condition precedes the subprogram's loop. This verification condition, which is not yet proved, guarantees that the loop *invariant* is preserved by the loop and that the invariant is sufficient to guarantee the loop's postcondition on termination of the loop. A proof for the verification condition has been begun. It includes three proof steps, each beginning with the word BY. The second step of the proof applies a general simplifier for arithmetic and predicate calculus, which reduces the statement to be proved to **true**. This fact is recorded in the step BY simplification.

The first proof step applies the theorem *fplus1* in the *Factorial trait* (shown in Figure 2.2) to simplify the verification condition by rewriting. (In this case Penelope was able to guess that the theorem should be instantiated with *ntemp* for m. Sometimes the user has to provide that information.) The instantiated theorem says that rewriting is possible, *if* $ntemp >= 0$, so this fact must be established. That's the job of the subproof introduced by establishing. The second subproof shows the simplified precondition.

The Penelope prover expresses what is left to prove as a *sequent*. In this example we see a sequent for the first subproof. The hypotheses in a sequent are numbered, beginning with 1. The conclusion follows, beginning with >>. The notation <proof> indicates an incomplete proof.

After the **loop** statement is the loop invariant, provided by the programmer (note the `--|`). The loop invariant is a generalization of the loop's postcondition, conjoined with the fact that $ntemp \geq 0$, which is needed in order to apply the *fplus1* theorem defined in Figure 2.2. In developing the loop, the programmer might first use a generalization of the postcondition as the invariant, strengthening it as required.

While we are verifying a program we can inspect the *precondition* of a statement (that is, the condition that must be true before the statement is executed) by clicking on the help-pane item `show-precondition`. We can inspect a *postcondition* (the precondition of the following statement) by clicking on `show-postcondition`. We can simplify preconditions by clicking on either `simplify-precondition` or `simplify-postcondition`.

Too many unproved sequents can clutter the Penelope screen. You can *hide* unproved sequents by clicking on the help-pane item `hide-sequent`. Hidden proofs are indicated by the symbol <> in the buffer.

You may have noticed that although the proof of `fact` appeals to theorem *fplus1* it nowhere mentions *f0*, which forms the basis of the recursive definition. Because its definition marked *f0* as a "rewrite" Penelope automatically applied *f0*, eliminating the need for the programmer to mention it explicitly. Also, the invariant does not mention that $ntemp <= n$. That fact, crucial for proving termination, is not needed for a proof of partial correctness.

2.4 Exiting Penelope and saving your work

Exiting Penelope and saving your work are two distinct steps. You can leave Penelope at any time by executing the `exit` command (on the `File` menu) or by entering Control-C (`^C`). To resume work later on a program and verification, you must save your work before exiting Penelope.

To save your work, issue the `write-named-file` command (`^X^W`). A pop-up window invites you to enter the file's name and format. Three formats are available: text, structure, and attributed. Text results in a small file containing the screen contents. Structure results in a larger file that you can't read but that Penelope can read in without parsing. Attributed results in a quite large file that Penelope can read in very quickly because it contains all of Penelope's internal information so that information doesn't have to be recomputed. One warning: only the text format can survive changes from one version of Penelope to the next. Before changing Penelope versions, you have to save all work in text format.

When saving in text format, make sure that no hidden verification condition is hiding work on a proof. Hidden verification conditions that are written out in text format and read in again are lost.[2] You can set the maximum length of lines in text files by invoking the `set-parameters` command on the `Option` menu and editing the `ABSOLUTE RIGHT MARGIN` parameter.

[2] In the structural and attributed formats they are saved even though you cannot see them.

If the current buffer is fully verified, you can enter it into the library, so that you can use it for further verification. See Section 2.5 on how to enter a file into the library.

2.5 The library

The result of verification with Penelope is a file that looks like Ada source code with many comments. In addition, if you execute the `write-library` command, Penelope produces a *library information file*. The information file for the Ada compilation unit `foo` is called `foo.lib`, and the information file for trait `bar` is called `bar.trait.lib`.

Library information files are kept in a library directory, named in a library annotation (see Section 6.5.1) at the head of the buffer.[3] All library information files for a given buffer must be in the named directory. The `write-library` command writes library information files for all compilation units in the Penelope buffer.

Note that writing to the library is a distinct step from writing out your Penelope buffer to a file. Maintaining consistency between the file of verified code and the library information file is currently the user's responsibility. In a future version of Penelope verified code will be kept in the library along with the library information file.

2.6 Status of the verification

At the head of each buffer, Penelope displays the verification status of the contents of the buffer. When performing a verification, you also need to know when the verification of a larger program, involving several separately verified modules, is complete. Penelope does not currently compute this status; you must do it yourself.

Assume that we have a current buffer containing a main program annotation. The buffer refers to a library, and Penelope was invoked from a directory containing a number of files with the results of earlier verifications. In order for the verification to be complete, the following must hold:

1. The verification condition for the main program annotation must be proved.
2. All verification conditions for all compilation units transitively "with'd" by the main program must be proved.
3. The mathematics of the specification for each compilation unit must satisfy all logical requirements (see Chapter 7).
4. Each compilation unit must be *complete* (no placeholders) and contain syntactically correct Ada code. Note that Penelope does not include complete syntactic checks. In

[3] This library directory must exist before you write to it.

addition, the code of each compilation unit must satisfy certain static requirements (see Appendix B).

5. If compilation unit `a` depends on compilation unit `b`, in the sense that `a` names `b` in a context clause, then one of the following must hold:
   - Compilation unit `a` was verified after `b`.
   - Changes made to `b` after the verification of `a` do not affect `a`.

Requirement 5 is similar to Ada requirements on compilation order (see [1, Sec. 10.3]). Ada compilers typically check that compilation units have been compiled in the correct order and note when compilation units are obsolete and must be recompiled. Unfortunately, Penelope does not yet provide such assistance. A future version of Penelope will provide mechanical support for the consistency checking implied by requirement 5.

Chapter 3

Lexical matters

3.1 Case sensitivity

Three different languages are used in the Penelope buffer: the Ada programming language, the Larch/Ada specification language for annotating Ada programs, and a dialect of the Larch Shared Language for developing mathematics. The Ada language is case-insensitive: upper- and lowercase letters may be used interchangeably in identifiers or keywords. The Larch Shared Language, on the other hand, is case-sensitive. Larch/Ada references both the objects of an Ada program and the functions defined in the Larch Shared Language, so it inherits the Larch Shared Language's case sensitivity. For example, `X` and `x` represent the same value in an Ada subprogram, but distinct values in Larch/Ada.

Penelope must provide Larch/Ada names for Ada objects. We have resolved this conflict in the following way. All Ada *simple names* are mapped to lowercase in Larch/Ada. For example, Ada `x` and `X` are both represented by Larch/Ada x. Larch/Ada X does not represent any Ada value.

3.2 Special characters

In addition to the Ada compound delimiters (see [1, Sec. 2.2]), Penelope defines the following compound delimiters:

```
--|     ->      <~~
--!     <=>     <!
--:     ~~      >~~
::      ~/~     >!
```

3.3 Identifiers and numeric literals

Identifiers, integers, and real literals appear in Larch/Ada and in the Ada subset accepted by Penelope. Note that no underscores may occur within integer or real literals.

Penelope may generate *system identifiers*, which are just like Ada identifiers, except that they end in an underscore. Such identifiers are used when Penelope must avoid conflicts with any identifiers possibly used in a program or theory.

The following lexemes are described using regular expressions as accepted by the Unix lexical analyzer `lex` [15].

⟨*identifier*⟩ ::= `[a-zA-Z][_a-zA-Z0-9]*`
⟨*system identifier*⟩ ::= `[a-zA-Z][_a-zA-Z0-9]*[_]`
⟨*integer*⟩ ::= `[0-9]+`
⟨*real literal*⟩ ::= ⟨*number*⟩ | ⟨*longnumber*⟩
⟨*number*⟩ ::= `[0-9]+[.][0-9]+`
⟨*longnumber*⟩ ::= `[0-9]+[.][0-9]+[E][+\-]?[0-9]+`

3.4 Comments

An Ada comment is a lexical construct that can be inserted anywhere in a program and is ignored by compilers and other applications that manipulate programs. Penelope does not support comments appearing at arbitrary places in the program text. A comment may be inserted where a declaration or a statement is expected. Penelope will read and discard comments appearing in other contexts.

The compound delimiters `--|`, `--!`, and `--:` introduce pseudo-comments with special meaning in Penelope.[1] Comments must not begin with these delimiters. To be properly parsed, Ada comments should begin with two hyphens followed by a blank space (`-- `).

3.5 Reserved words

All Ada reserved words are reserved in Penelope (see [1, Sec. 2.9]). In addition, the identifiers of Table 3.1 are reserved for use in Larch/Ada and cannot be used in Ada programs to be verified.

[1]The delimiter `--:` is not currently used but is reserved for future use.

assert	invariant	such
conclusion	lemma	that
exists	occurrence	trait
false	precondition	true
forall	promise	virtual
global	spec	where

Table 3.1: Larch/Ada reserved words

Chapter 4

Ada types and mathematical sorts

In the design of Penelope, we have adopted the Larch [10, 11] approach to specification in choosing to separate a specification into two parts, a mathematical part and an interface part. The mathematical part defines sets of values (called *sorts*) and operations on them. For example, the mathematical part of every specification automatically contains the sort *Int*, denoting the infinite set of mathematical integers, along with the usual arithmetic operations on it. We will often call the symbols introduced by the mathematical part "mathematical operations," in order to distinguish them emphatically from the executable operations of Ada. The interface part of the specification uses these sorts and serves as an interface between the "ideal world" of the mathematical part (in which there are no side-effects, exceptions, or resource limitations) and the more complex world of program behavior.

What underlies this interface is that each Ada type is associated with a unique sort, on which the type is said to be *based*. For example, the type `integer` is based on sort *Int*. By saying this we mean that any possible value of type `integer` is modeled as one of the values of sort *Int*, although the converse need not be true: infinitely many values of sort *Int* are not values of `integer`.

Let's look again at how we might specify the procedure

```
procedure prime_flag(x: in integer; b: out boolean);
```

which (provided `x` is not too big) should set `b` to true if `x` is prime and to false otherwise. We first introduce the mathematical operation

$$is\_prime : Int \rightarrow Bool$$

defining the meaning of "prime" for any mathematical integer. (We must provide axioms to define the meaning.) The set of values of the predefined sort *Bool* is **true** and **false**. Type

`boolean` is based on sort *Bool*.[1] We then specify `prime_flag`:

```
procedure prime_flag(x: in integer; b: out boolean);
  --| where
  --|   in 0 <= x and x <= 1000;
  --|   out b = is_prime(x);
  --| end where;
```

This specification says that, on entry, the value of `x` must lie between 0 and 1000 and on normal exit the value of `b` should be the value of *is_prime(x)*. (Note that the last expression is of sort *Bool*, on which the type of `b` is based.) As previously indicated, this annotation asserts by default that execution of `prime_flag` does not propagate any exceptions. Although it is meaningless to speak of "resource constraints" on mathematical functions such as *is_prime*, the possibility of such constraints on `prime_flag` (for example, an array of finite length may be used in the computation) is recognized by restricting calls of the procedure to those in which the actual `x` parameter is relatively small. It is important to realize that in this specification every symbol other than x and b is either a keyword of the specification language, a mathematical constant, or a mathematical operator. Symbols for executable operations *never* occur in specification constructs. So we could have used the name "prime_flag" for both the Ada procedure and the mathematical operation.

The mathematical part of a specification consists of a set of declarations introducing sorts and mathematical operations on them, and a set of axioms defining those operations. Some of the declarations and axioms are supplied automatically by Penelope (such as the sort of integers and the usual arithmetic operations on them) and others are supplied explicitly by the user. Penelope provides a way of entering the mathematical part of a specification, thus making it available to the prover and simplifier.

4.1 Sorts and types

Penelope includes support for sorts corresponding to Ada's built-in data types and type constructors. A library of examples distributed with Penelope includes other sorts that may be of use in specifying and verifying new programs. We can also use the Larch Shared Language to introduce new sort names and define functions on the sorts (see Chapter 7).

Each sort has a name, called a *sortmark*. A sortmark may be an identifier or a structured sortmark. Structured sortmarks are used for sorts corresponding to structured Ada types (enumeration, array, and record sorts).

⟨*sortmark*⟩ ::= ⟨*identifier*⟩

[1] In fact *Bool* is distinct from the enumeration sort *AdaBool*, on which the Ada predefined type `boolean` is properly based. Operations like *pred* and *succ* are defined for the sort *AdaBool*, but not for *Bool*. In the current implementation, Ada boolean types are based on terms of sort *Bool*. Sort *AdaBool* will be implemented in a later version.

Structured sortmarks are a Penelope extension to the Larch Shared Language.

Note that sorts correspond to Ada types and *not* to Ada subtypes. Hence, for example, Ada subtypes `positive` and `-1..4` are both based on sort *Int*, because they are subtypes of type `integer`.

In our model structurally similar Ada types are based on the same sort, even though Ada's type checking uses name equivalence. Thus, given the Ada declarations

```
type a is array (1..10) of integer;
type b is array (1..10) of integer;
```

types `a` and `b` are distinct, but our model bases both on the same sort, which is also the sort on which type `c` is based:

```
type c is array (0..100) of integer;
```

Structural equivalence simplifies the mathematics of specification by identifying isomorphic sorts, but it means that there is not a one-to-one correspondence between Ada types and Larch/Ada sorts.

4.2 The sort *Int*

All Ada integer types are based on the predefined sort *Int*, which consists of the infinite integers with the usual mathematical operations.

The Larch/Ada operators associated with integer division follow Ada conventions rather than the usual mathematical conventions. Ada integer division rounds toward zero. Thus, $(-a)/b = -(a/b)$.[2] Also, **rem** and **mod** are distinct and not always positive: a **mod** b has the sign of b, and a **rem** b has the sign of a. (See [1, Sec. 4.5.5].)

4.3 Operations on discrete sorts

Discrete sorts include the integers and enumeration sorts (see Section 4.9). The mathematical operations on discrete sorts currently supported in Penelope are the relational operators `=`, `/=`, `>`, `>=`, `<`, and `<=`. In addition, Table 4.1 shows operations on S corresponding to Ada operations on T, where S is the sort of objects in T.[3]

[2] To mathematicians, $(-5)/2 = \lfloor -2.5 \rfloor = -3$. In Ada and Larch/Ada, it's $-\lfloor 2.5 \rfloor = -2$.
[3] They are defined in [8].

Operator	Signature	Meaning
`first`	$\rightarrow S$	T'BASE'FIRST
`last`	$\rightarrow S$	T'BASE'LAST
`succ`	$S \rightarrow S$	T'SUCC
`pred`	$S \rightarrow S$	T'PRED
`pos`	$S \rightarrow Int$	T'POS
`val`	$Int \rightarrow S$	T'VAL

Table 4.1: Operations on discrete sorts

4.4 The sort *Real*

It is well known that real number arithmetic on computers, and hence in languages such as Ada, does not have the same properties as real number arithmetic in mathematics. Most real numbers cannot even be represented exactly in computers. Of course, most integers cannot be represented either, but in the case of integers, we can at least define an interval within which all integers are represented. If we can convince ourselves that our computation lies within that interval, we are sure that the values we are interested in are accurately represented in the computer. By contrast, in real number computations it is usually the case that the values we are interested in (like π) are not represented accurately in any computer.

In Penelope, specifications about floating point programs assert that they are *asymptotically correct*. Each real number[4] is represented in a computer by some rational. We say that two real numbers are "approximately equal" or "infinitely close" (written $\sim\sim$ to resemble the mathematical symbol $\approx$) if their representations tend to equality as the representation of the reals becomes more and more accurate. Thus if s is the result of an algorithm computing the sine of θ by a series, we assert the asymptotic correctness of the algorithm by requiring $s \sim\sim \sin\theta$. That is, our algorithm is asymptotically correct if, *in the limit* as the computer representation of the reals becomes more accurate, the computed value of s approaches $\sin\theta$–although on any finite computer we can compute only an approximation. We do not normally use the symbol $=$ in specifying a sine function, because that would imply that our program produces the exact real number that is $\sin\theta$, which is not in general true. For this reason, although the ordinary relational operators are also defined on real numbers, we normally use the special symbols in Table 4.2 instead.

For a continuous function f, $f(\vec{x}) \approx f(\vec{y})$ if $\vec{x} \approx \vec{y}$. Exploiting this property is critical to verifying many computations on real numbers. If a mathematical function used in specifying a program is continuous, we can (and should) indicate that fact when we define the function (see Section 7.5). The `approximate-simplify` proof rule (see page 77) uses this information in simplifying terms involving reals.

When we use the operators above to specify programs, we can verify that *on a sufficiently accurate computer* our program will produce the desired result. Occasionally we need to

[4] For mathematicians, Penelope's sort *Real* consists of the limited nonstandard reals. If the operators $\sim\sim$ et. al. are removed, Penelope's real numbers are the normal reals of analysis.

Operator	Meaning
~~	Approximately equal ($\approx$)
~ / ~	Not approximately equal
<!	Strictly less than (not approximately equal)
<~~	Less than or approximately equal
>!	Strictly greater than (not approximately equal)
>~~	Greater than or approximately equal

Table 4.2: Approximate relational operators for the reals

specify a program using the more familiar operators (=, etc.), for example to show that an error quantity is ≥ 0, not just >~~ 0 (in which case it might be very slightly negative). Penelope supports this kind of specification, too, but proofs of such statements are much more difficult and should be attempted only by experts.

The Ada model of real arithmetic presents us with a second problem: operations on real numbers are, in general, non-deterministic. If x and y are real numbers, then the product of x and y may be any one of many rationals that approximate the mathematical product xy. How good the approximation is depends on the accuracy of intermediate representations on the machine. Penelope uses special predefined relations to represent the result of machine real operations. For example, *ftimes*(x, y, z) states that z is a possible result of multiplying x and y. (There are many real numbers z for which this may be true.) Note that *fequals*$(x, y, \mathbf{true})$ and *fequals*$(x, y, \mathbf{false})$ may both be true. If $x - y$ is small, the result of a computer equality test depends on the accuracy of the machine.

Table 4.3 summarizes the intuitive meaning of Larch/Ada's predefined functions for describing the results of Ada comparison operators on reals x and y. For each of the following, the first two operands are numeric and the third is boolean.[5]

Function	Meaning
fequals(x,y,b)	The test `x=y` may have boolean result `b`.
fne(x,y,b)	The test `x/=y` may have boolean result `b`.
fless(x,y,b)	The test `x<y` may have boolean result `b`.
fle(x,y,b)	The test `x<=y` may have boolean result `b`.
fgt(x,y,b)	The test `x>y` may have boolean result `b`.
fge(x,y,b)	The test `x>=y` may have boolean result `b`.

Table 4.3: Larch/Ada functions for Ada comparison operators on reals

Larch/Ada's predefined functions for describing the results of Ada arithmetic operators on floats are summarized in Table 4.4. They take three real numbers and return a boolean.

Note that although Penelope uses "f-predicates" (*fequals*, *fplus*, etc.) to represent the result of computer operations, we do not normally use them in specifications because in general we

[5] All of the operators describing the results of computer arithmetic and comparisons are defined for integers as well as for reals, although in practice they are used only for reals.

Function	Meaning
fplus(x,y,z)	Computer operation `x+y` may produce `z`.
fminus(x,y,z)	Computer operation `x-y` may produce `z`.
ftimes(x,y,z)	Computer operation `x*y` may produce `z`.
fdiv(x,y,z)	Computer operation `x/y` may produce `z`.

Table 4.4: Larch/Ada functions for Ada arithmetic operators on reals

cannot say anything about the results of computer operations.

Penelope automatically applies the axiom that if *fplus*(x, y, z) then $x + y \sim\sim z$[6], and automatically applies analogous axioms for the other f-predicates. That is one way of saying that, for example, the result of a machine computation approximates the corresponding mathematical result. However, it is possible to give a more precise description of the f-predicates in terms of *rounding*.

Function	Meaning
round_up(x)	Represents `x` rounded up.
round_down(x)	Represents `x` rounded down.

Table 4.5: Larch/Ada functions to describe rounding

If we ask Penelope to explicitly represent the effects of rounding, then the f-functions will be replaced by their definitions in terms of the rounding functions of Table 4.5. For example, *fplus(x,y,z)* becomes

round_down(x)+round_down(y) $<=$ z **and**
z $<=$ *round_up(x)+round_up(y)*

The advantage is this: According to the Ada model, *some* real numbers, called *safe* numbers, *are* represented exactly; for example, when the mathematical sum of two safe numbers is also safe, the machine sum is required to return that mathematical value. Penelope assumes that all the dyadic rationals (rationals whose denominator, represented in least terms, is a power of two) are safe and expresses that by the axiom that a safe number rounds to itself. By representing rounding explicitly we therefore obtain some extra proving power.

4.5 The sort *Bool*

The predefined sort *Bool* consists of the values **true** and **false**, with the usual mathematical operations.

[6]The converse is not true.

4.6 Map sorts

A Larch/Ada map is what is usually called a *map* or *array* in mathematics. We choose the word "map" in order to avoid confusion with the rather different semantics of Ada arrays (see below). Map operators are component indexing, component replacement, and quantified component replacement (see Section 5.8).

$$\langle \mathit{sortmark} \rangle ::= \\ \textbf{map}[\ [[\langle \mathit{sortmark} \rangle]]^{+}_{,}\ ]\ \textbf{of}\ \langle \mathit{sortmark} \rangle$$

Maps must be declared (in a sort declaration—see Section 7.3) before use.

4.7 Tuple sorts

A Larch/Ada tuple is what is usually called a *tuple* or *record* in mathematics. We choose the word "tuple" in order to avoid confusion with the rather different semantics of Ada records (see below). Tuple operators are component selection and component replacement. In addition, we can directly represent tuple values (see Section 5.9).

$$\langle \mathit{sortmark} \rangle ::= \\ \textbf{tuple of}\ [[\langle \mathit{fieldsortmark} \rangle]]^{+}_{,}$$

Tuples must be declared (in a sort declaration—see Section 7.3) before use.

4.8 Array and record sorts

Like `integer` and `real`, Ada composite types are also based on sorts. But although there is just one sort *Int*, there is a different "array sort" for each possible combination of index sorts and component sort. For example, if in Penelope a type is declared as **array** `(integer)` **of** `integer`,[7] then the type is based on the array sort **array**`[`*Int*`]` **of** `[`*Int*`]`. Array and record sorts differ from map and tuple sorts, respectively, in that we are concerned about *definedness* for Ada arrays and records. An Ada array may be viewed as a tuple together with a boolean map (over the same indices) indicating whether each component of the Ada array is defined or not. Similarly records differ from tuples because we are concerned about definedness, and also about variants.[8]

Array and record operators are defined for array and record sorts, respectively. Array operators are component indexing and component replacement (see Section 5.8). Record operators are component selection and component replacement (see Section 5.9).

[7] Penelope does not yet support subtypes such as `1..10`. In the current version it is necessary to use the type `integer` for array indices in these declarations.

[8] Definedness and record variants will be implemented in a future version of Penelope.

$\langle sortmark \rangle$ ::=
  array[[[$\langle sortmark \rangle$]]$^{+}_{,}$] **of** [$\langle sortmark \rangle$]
  | **record** [[$\langle fieldsortmark \rangle$]]$^{+}_{;}$ **end record**
$\langle fieldsortmark \rangle$ ::=
  $\langle identifier \rangle$: $\langle sortmark \rangle$

AnyArraySort is the mathematical domain of all arrays, with component indexing and replacement. Similarly, the sort *AnyRecordSort* is the mathematical domain of all records, with component selection and replacement.

Note that because Penelope supports structural equivalence of array and record sorts, two distinct Ada types may be based on the same sort.

4.9 Enumeration sorts

Enumeration sorts correspond to Ada's enumeration types. The enumeration sort *(red, yellow, blue)* includes the enumeration literals *red*, *yellow*, and *blue*, in that order. The sort also includes an infinite number of other elements, for example, *succ(blue), succ(succ(blue)), pred(red)*, etc.

$\langle sortmark \rangle$ ::=
  | ([[$\langle enumeration\ literal \rangle$]]$^{+}_{,}$)

$\langle enumeration\ literal \rangle$ ::=
  $\langle identifier\ literal \rangle$
  | $\langle character\ literal \rangle$
$\langle identifier\ literal \rangle$ ::= $\langle function\ name \rangle$
$\langle character\ literal \rangle$ ::=
  ' $\langle character\ representation \rangle$ '
$\langle character\ representation \rangle$ ::=
  $\langle printable\ character \rangle$
  | '\$\langle number\ from\ 0\ to\ 127 \rangle$'

The lexical rules of Larch/Ada are such that the Ada enumeration literal `red`, which can also be written `RED`, is always written in small letters in Larch/Ada: *red*.

An enumeration literal is a Larch/Ada constant and is represented as such. That is, it consists of an identifier or character literal and a signature (see Section 7.5). When no ambiguity is possible, we use the identifier or character literal alone, but when necessary we

add the signature. Thus if *red* is an enumeration literal of sort *color*, we can write *red* or *red()* or *(red:->color)*.[9]

Character literals may be graphic characters or ASCII octal representations of characters. The sort *Char* is predefined. For example, `'a'` and `'\068'` are members of sort *Char*.

In the current implementation of Penelope, Ada boolean types are based on sort *Bool*, rather than on an enumeration sort.

[9] It is also correct to write *(red:->color)()*, but there is no excuse for it.

Chapter 5

Terms

The Ada language is designed for computation, not for reasoning. We use Ada expressions to instruct a machine about what computations to perform. We use Larch/Ada *terms* to denote the possible results of such computations, the values on which Ada objects are based. Such values include constants (such as 1, 2, **true**, etc.) and the result of applying operators to terms (e.g., $1 + x$, **not** p, *min(x,y)*). These terms are a part of the Larch/Ada language. They are defined by and appear in the mathematical part of specifications (see Chapter 7). In this chapter we define the syntax of terms and informally describe their meaning.

Larch/Ada is a *sorted language*. Just as each expression in Ada has a type, so each term in Larch/Ada has a sort. (We follow Larch in using the word *type* for program values and *sort* for mathematical values.) For example, x + *true* is not an admissible term in the language, because + is not defined for an argument of sort *Int* and an argument of sort *Bool.*

A note on notation: terms are mathematical expressions and can be written using either mathematical notation or Larch/Ada syntax. Thus we can write "$p \wedge q$" or "p **and** q". In this manual we will almost always use symbols corresponding to what we see on the Penelope screen, although occasionally we will resort to standard mathematical notation when we are not explicitly referring to something we might write in a specification.[1]

[1] Following the default used by Penelope, we present Ada code in a typewriter font (e.g., `z:= x + y;`) and specification in italics (e.g., $z = x + y$). The font used in this manual or in Penelope has no formal significance.

The syntax of terms is

⟨*term*⟩ ::=
  true
| **false**
| ⟨*integer*⟩
| ⟨*real literal*⟩
| ⟨*sortmarked_variable*⟩
| ⟨*ada_variable*⟩
| ⟨*simple_id*⟩
| ⟨*unary operator*⟩ ⟨*term*⟩
| ⟨*term*⟩ ⟨*binary operator*⟩ ⟨*term*⟩
| ⟨*function application*⟩
| ⟨*modified term*⟩
| ⟨*conditional term*⟩
| ⟨*bound term*⟩
| ⟨*array term*⟩
| ⟨*record term*⟩
| ⟨*aggregate*⟩
| ⟨*character literal*⟩

5.1 Constants

The boolean values **true** and **false**, as well as integer and real constants, are predefined in Larch/Ada.

5.2 Variables

Larch/Ada, like Ada, supports a rich universe of entities that can be referred to by name. The Ada *simple name* **x** might refer to a package, an Ada object, or an enumeration literal. Larch/Ada supports the same kinds of entities, but also logical variables, for example, the bound variable x in

forall $x{:}Int :: x+0=x$

Such variables are indispensable in writing specifications. Larch/Ada therefore generalizes the (already rather large) Ada concept of *simple name* to *variable*.[2]

⟨*term*⟩ ::=
  | ⟨*simple_id*⟩
  | ⟨*ada_variable*⟩
  | ⟨*sortmarked_variable*⟩
⟨*simple_id*⟩ ::= [[⟨*qualifier*⟩]]* ⟨*identifier*⟩
⟨*ada_variable*⟩ ::=
  ("⟨*declarative_context*⟩":⟨*identifier*⟩)
⟨*declarative_context*⟩ ::=
  [[⟨*declarative_region_name*⟩]]+.
⟨*declarative_region_name*⟩ ::=
  an identifier optionally followed by a parameter-result profile
⟨*sortmarked_variable*⟩ ::=
  ⟨*identifier*⟩: ⟨*sortmark*⟩

The simplest form of variable is an identifier. In order to avoid confusion, it is a good idea not to use the same identifier for distinct logical variables and Ada names. Nevertheless, it can happen that the same name comes to be used in more than one way in a program. In Larch/Ada, the denotation of a variable that is a simple identifier is resolved to be the first possible one of the following:

bound variable An identifier resolves to a bound variable if the identifier is bound by a quantifier or by a variable list introducing axioms or lemmas (see Chapter 7).

Ada value An identifier may denote the current Ada definition of the identifier, if any. This definition may be an Ada object or enumeration literal. By the rules of Ada overload resolution, an enumeration literal is denoted only if the type of that literal is compatible with the sort required by the context in which the identifier occurs. For example, even if the enumeration literal `red` is visible at the point where the term *red = 1* occurs, the identifier *red* is *not* interpreted as an enumeration literal, since the context requires an integer.

Larch enumeration literal We can define enumeration literals and enumeration sorts in the Larch Shared Language. See the discussion of enumeration literals (Section 4.9).

Larch constant We can define constants (e.g., the *empty* stack) in the Larch Shared Language. See Section 5.4.

free variable An identifier may refer to a free mathematical variable, as for example, in $x = x + 0$. Free variables are allowed only in axioms, lemmas, and proofs.

[2] In the future Penelope may abandon the term "variable" in favor of some variant on "name," as Ada has done.

Sometimes we need to refer to an Ada variable that is not visible, perhaps because it is hidden by another declaration of the same name. Larch/Ada syntax enables us to refer to Ada variables and logical variables unambiguously by providing *Ada variables* and *sortmarked variables.*

Ada entities are unambiguously denoted by a *declarative context* and an identifier. The combination uniquely identifies the declaration of the identifier in the given declarative region. The declarative context is a sequence of names of declarative regions separated by periods, beginning with `standard`, and ending with the name of the region in which the object was declared, for example, `standard.p.q`. The declarative context is similar to the Ada prefix of an expanded name (see [1], Section 4.1.3), except that it must be complete and that subprogram names are modified by their parameter-result profiles. Although unambiguous names for Ada entities are sometimes generated by Penelope, you should not use them in annotations. It is usually possible to avoid situations in which you would have to write such a name.

A Larch/Ada variable is unambiguously represented by a *sortmarked variable* (i.e., an identifier tagged with the name of a sort). For example, the logical variable *x:Int* has identifier *x* and sort *Int.*

The unambiguous representation of enumeration literals is discussed in Section 4.9.

5.3 Unary and binary operators

⟨*unary operator*⟩ ::=
  + | − | **abs** | **not**
⟨*binary operator*⟩ ::=
  and | **or** | **xor** | ->

  | = | /= | < | <= | > | >=
  | + | − | & | * | / | **mod** | **rem** | **
  | ⟨*real operator*⟩
⟨*real operator*⟩ ::=
  | ~~ | ~/~ | >! | >~~ | <! | <~~

Larch/Ada operators are defined corresponding to most of Ada's unary and binary operators on integers, reals, and booleans. It is important to remember that they refer to total mathematical functions and never "fail" or raise exceptions. In cases where an Ada expression raises an exception, the corresponding Larch/Ada term still denotes a value (e.g., $x/0$) for which there is no corresponding value in the Ada type. Examples of unary and binary operators in terms are $a + b$, $x = 6$, or $x > 7$. Note that no operators of the term language correspond to the Ada short circuit control forms **and then** and **or else**, since these do not merely express a value but may change the flow of control of the program.

In this manual we ordinarily represent Larch/Ada operators as they appear in Penelope, for

example, /= rather than the mathematical symbol $\neq$ and **and** rather than the mathematical symbol $\wedge$. The two forms are, however, equivalent, and this manual may occasionally use the mathematical symbol.

Table 5.1 summarizes the associativity and precedence of the supported predefined operators. The operators are given in order of increasing precedence.

Operator	Associativity
->	left
and, or, xor	left
<, <=, >, >=, =, /=	none
~~, ~/~, >!, >~~, <!, <~~	none
+, −, &	left
unary +, unary −,	none
*, /, **mod, rem**	left
not, abs, **	none

Table 5.1: Larch/Ada operator precedence

In most cases the meaning of the operator is the usual, well-understood mathematical operation, but see Section 4.2 on integer division and related operations. Real numbers and operations on reals are described in Section 4.4.

5.4 Function application

$\langle\textit{function application}\rangle$ ::=
  $\langle\textit{function\_name}\rangle\,([[\langle\textit{term}\rangle]]^*_,)$
$\langle\textit{function\_name}\rangle$ ::=
  $\langle\textit{identifier}\rangle$
  | $(\langle\textit{identifier}\rangle : \langle\textit{signature}\rangle)$

The user may define mathematical functions and apply them to arguments. *The function name never refers to an Ada function.* Thus the same identifier may be used for an Ada function and a mathematical function without ambiguity. It may be preferable at times to choose distinct names for mathematical functions in order to avoid confusion for the human reader.

Function names may be overloaded. When the name (identifier) of a function is overloaded, the result sort may be appended to the function application to distinguish the two functions. For example, the function application *is_prime(x)* may also be written *is_prime(x):Bool.*

Mathematically, a constant is a function with no arguments. Thus the integer constant c may also be written *(c:Int)*. The latter form is used to avoid confusion when the symbol c is

overloaded. Empty parentheses after constants are optional, but may be useful to distinguish a constant from some other symbol using the same identifier—for example, a variable. So we can also write $c()$ or $c() : Int$.

Some predefined functions use the function application syntax (see, for example, Section 4.4).

5.5 Two-state terms

A *state* σ is a function that associates a value with every program object. We often use the terminology "the value of a variable (or object) in state σ." States are important because the effects of executing an Ada program can be described by describing the concomitant changes in the values of program objects, that is, the changes in state.

The notion of state can be extended so that a state σ associates a value to every term. The value has the same sort as the term. If a term contains Ada variables denoting program objects, the only way we can figure out what value that term denotes is to apply a state to it.

Three different states are of interest in the annotation of an Ada program.

entry A subprogram annotation also makes assumptions about values of Ada objects on entry to the subprogram.

exit A subprogram annotation makes claims about the values of Ada objects on exit from the subprogram.

current Other annotations (embedded assertions, loop invariants, etc.) may make claims about the values of objects in the current state (i.e., the state at that point in the program).

It may happen that in an exit annotation we wish to refer to the value of a variable on exit from and also on entry to the subprogram, for example, to say that the subprogram increments the entry value. The reserved word **in** designates the value of a variable or term in the entry state.

⟨*modified term*⟩ ::=
  in ⟨*variable*⟩
| **in** ⟨*term*⟩

To specify a sort subprogram, for example, we might write

```
procedure sort_array (a: in out intarray);
  --| where
```

```
--|    out (permutation(a, in a) and sorted(a));
--| end where;
```

where *permutation* and *sorted* are mathematical functions on arrays.

5.6 Conditional terms

⟨*term*⟩ ::=
  if ⟨*term*⟩ **then** ⟨*term*⟩ **else** ⟨*term*⟩

The last two subterms must be of the same sort, and the first subterm must be boolean. If q and r are boolean terms, then the term **if** p **then** q **else** r is equivalent to $(p \wedge q) \vee (\neg p \wedge r)$.

The following example shows a boolean conditional term and also an integer conditional term.

```
function abs_max (a,b:  in integer) return integer;
  --| where
  --|   in if a>=0 then b>=0 else b<0;
  --|   return if a>0 then max(a,b) else max(-a,-b);
  --|  end where;
```

5.7 Bound terms

⟨*term*⟩ ::=
  ⟨*quantifier*⟩ ⟨*varlist*⟩ : : ⟨*term*⟩
⟨*quantifier*⟩ ::= **forall** | **exists** | **lambda**
⟨*varlist*⟩ ::= $[[\langle identifier\rangle[[:\ \langle sortmark\rangle]]\ ]]^{+}_{,}$

The subterm of a quantified term (with **exists** or **forall**) must be boolean.

For example:

```
forall i:Int::i>=0 and i<n -> a[z]>=a[i];
```

The variables in the ⟨*varlist*⟩ must all have distinct identifiers. If a list of identifiers is followed by a sortmark, all identifiers get the same sortmark. For example, *x,y,z:Int* is equivalent to *x:Int,y:Int,z:Int*. The last variable in the list must have a sortmark. In the subterm, variables bound by the quantifier do not have to be sortmarked.

5.8 Array and map terms

$\langle$*array terms*$\rangle$ ::=
  $\langle$*term*$\rangle$ [[[$\langle$*term*$\rangle$]]$^{+}_{,}$]
  | $\langle$*term*$\rangle$ [[[$\langle$*term*$\rangle$]]$^{+}_{,}$ => $\langle$*term*$\rangle$]

If a is a two-dimensional array or map, then $a[i,j]$ represents the value of a component of a. Note that in Larch/Ada we use square brackets for components of arrays, rather than parentheses as in Ada. It is often useful to represent the value of a if a component is replaced. Suppose we replace $a[i,j]$ with v. We represent the resulting array value by *a[i,j => v]*.

5.9 Record terms, tuple terms, and expanded names

$\langle$*record term*$\rangle$ ::=
  $\langle$*term*$\rangle$. $\langle$*selector*$\rangle$
  | $\langle$*term*$\rangle$ [. $\langle$*selector*$\rangle$ => $\langle$*term*$\rangle$]
$\langle$*selector*$\rangle$ ::=
$\langle$*identifier*$\rangle$
  | $\langle$*ada_operator_symbol*$\rangle$

If r is a record or tuple then *r.f* represents field f of r. It is often useful to represent the value of r if a component is replaced. Suppose we replace component f of r with v. We can represent the resulting record value by *r[.f=>v]*.

The term *r.f* may also be an *expanded name*. If r is a package, then *r.f* denotes object f declared in package r. Component replacement for packages is meaningless and may not appear in terms. As in Ada, an expanded name may refer to a subprogram (defined in a package) whose name is an operator.

5.10 Aggregates

$\langle$*aggregate*$\rangle$ ::=
  [[[$\langle$*identifier*$\rangle$: $\langle$*term*$\rangle$]]$^{+}_{,}$]

We can represent a particular tuple value by providing values for all of the fields in the tuple. The $\langle$*identifier*$\rangle$s must represent field names in a declared tuple sort and all field names for the sort must be present.

5.11 Ill-formed input

We sometimes enter a program or specification that is ill-formed and does not type-check or sort-check. When Penelope encounters improper input, it does not produce preconditions

or verification conditions. Instead, the flag **undefined** appears. Note that **undefined** does not denote a value in any semantic domain. It simply means that the input to Penelope is ill-formed and Penelope cannot produce meaningful results.

Chapter 6

Larch/Ada: Specifying Ada programs

In this chapter we describe how we can specify an Ada program by *annotations*. A Larch-style specification is two-tiered: the mathematical tier defines functions and provides theorems underlying the program. The interface tier uses these definitions to specify what the program should do. The interface tier is written in a *Larch interface language*, of which Larch/Ada is an example. A variant of the Larch Shared Language—described in Chapter 7—is used to define the mathematical tier. In Larch/Ada we specify Ada programs by annotating them with terms (see Chapter 5) that represent Ada program objects and assertions about them. Sometimes we will informally refer to the annotations as the specification of the program.

Penelope follows a familiar model of specification. We annotate the program with entry and exit conditions. Penelope computes a weakest precondition for the program, given our exit condition, and we must show that the entry condition implies the precondition. The mathematical statement of that implication is called a verification condition.

Most programs are broken up into modules, and a Penelope verification typically breaks each Ada module into yet-smaller proof units associated with their own verification conditions. These units include subprograms and **loop** statements. Other annotations identify the mathematical part of a specification or support abstraction. Larch/Ada annotations all begin with the compound delimiter `--|`.

Most of our specification of a program consists of assertions that should hold in particular states of the program. An *assertion* is a term of sort *Bool.*

$$\langle assertion \rangle ::= \quad \langle term \rangle$$

For each kind of annotation we will say which state of the program it refers to.

6.1 Subprogram annotations

A *subprogram annotation* represents a contract between the subprogram and its callers. The subprogram annotation states what must be true when the subprogram is called (the responsibility of the caller) and what is then guaranteed to be true if the subprogram terminates. Externally, every caller must show that the entry conditions of the subprogram are satisfied at the point of the call, and the caller may assume that the exit conditions hold if the subprogram returns. Internally, the implementor must show that if the entry conditions hold and the program terminates then the precondition of the exit condition holds.

Recall that in a *two-state predicate*, subterms of the assertion may be modified by **in**, and such subterms get their values from the entry state. Other subterms get their values from the current or exit state. In a subprogram annotation using a two-state predicate, the entry state is the state on entry to the subprogram, and the exit state is the state on termination (which may be normal or exceptional according to the annotation).

Note that in the above discussion we do not assume that the subprogram must terminate. That is, the subprogram annotation specifies conditions for the *partial correctness* of the subprogram, as opposed to *total correctness*, which additionally requires that the program terminate.

6.2 Syntax of subprogram annotations

Subprogram annotations may follow subprogram declarations:

⟨*subprogram declaration*⟩ ::=
  ⟨*subprogram_spec*⟩;
  ⟨*subprogram annotation*⟩

Or they may precede the reserved word **is** in a subprogram body:

⟨*subprogram body*⟩ ::=
  ⟨*subprogram_spec*⟩
  ⟨*subprogram annotation*⟩
  is ⟨*body*⟩

The default annotation of a subprogram body is the annotation of the subprogram's declaration, if there is a separate declaration.

The syntax of the ⟨*subprogram annotation*⟩ is:

⟨*subprogram annotation*⟩ ::=
  --| where

[[⟨*side effect annotation*⟩]]*
[[⟨*in annotation*⟩]]*
[[⟨*out annotation*⟩]]*
[[⟨*result annotation*⟩]]*
[[⟨*propagation constraint*⟩]]*
[[⟨*propagation promise*⟩]]*
--| **end where**;

6.2.1 Side effect annotations

⟨*side effect annotation*⟩ ::=
--| **global** ⟨*variable parameters*⟩ ;
⟨*variable parameters*⟩ ::=
$[[\ [[\langle \mathit{term} \rangle]]^{+}_{,} : \langle \mathit{mode} \rangle]]^{+}_{;}$

The ⟨*variable parameters*⟩ list the global objects potentially read and written by this subprogram. For example, suppose we wish to implement a stack package for a particular stack, called `my_stack`. We assume that a trait (see Figure A.1 on page 96) provides us with functions *top* and *pop* that denote the top element of the stack and the rest of the stack respectively. We may wish to implement a `pop_stack` function in which the top element is removed from the stack and returned to the caller. Thus there is a side effect on `my_stack`.

```
function pop_stack return integer;
  --| where
  --|    global my_stack: in out;
  --|    out my_stack=pop(in my_stack);
  --|    return top(in my_stack);
  --| end where;
```

The parameter names appearing in the ⟨*variable parameters*⟩ are the names of visible global objects (possibly extended names). They are called the *global parameters* or sometimes the *implicit parameters* of the subprogram.

The side effect annotation of a subprogram must list all objects read or written by the program, or any subprogram that it (transitively) calls. That is, if subprogram `a` calls subprogram `b`, which may modify object `v`, then `v` must appear in the side effect annotation of `a` as well as in that of `b`. Omitting the side effect annotation is equivalent to specifying that the subprogram has no global parameters. A global variable may occur at most once in the side effect annotations for a program.

Note that global objects must appear in the side effect annotation if they are *potentially* read or written by the program. "Potentially" here means that they appear in a syntactic

construct implying reading or writing. For example, in the Ada statement **if false then** `x:=0` **end if;** the object **x** is considered to be potentially written, even though no execution of the statement will actually result in writing to **x**.

If a subprogram has distinct declaration and body, and if the annotations for the two differ, then every readable global parameter (mode **in** or **in out**) of the body must be a readable global parameter of the declaration and every writable global parameter of the body (mode **out** or **in out**) must be a writable global parameter of the declaration. It is good practice to declare every global parameter of the body a global parameter of the declaration, and to give the same mode in each declaration.

6.2.2 In annotations

⟨*in annotation*⟩ ::= --| **in** ⟨*assertion*⟩;

where the assertion is not a two-state predicate. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes **in** or **in out**.[1] If the ⟨*in annotation*⟩ is omitted, that is equivalent to an ⟨*in annotation*⟩ with an assertion of **true**.

The implementor is allowed to assume that, on entry to the subprogram, the state satisfies the assertion. Users of the subprogram must show that the state immediately preceding any call satisfies the assertion (when the values of the appropriate actual parameters are substituted for the formal parameters).

6.2.3 Out annotations

⟨*out annotation*⟩ ::= --| **out** ⟨*assertion*⟩;

where the assertion is a two-state predicate. Unless modified by **in**, all variables refer to the exit state. The only Ada variables allowed to appear in the assertion are those appearing in the formal part of the subprogram declaration and the subprogram's side effect annotations. If the ⟨*out annotation*⟩ is omitted, that is equivalent to an ⟨*out annotation*⟩ with an assertion of **true**.

Verification conditions for the subprogram are generated whose truth will guarantee that, if the subprogram is called in a state satisfying the ⟨*in annotation*⟩, and if it terminates normally (i.e., without propagating an exception), the state after termination will satisfy the ⟨*out annotation*⟩.

[1] This is a simplification. In principle Penelope also allows some attributes of formal parameters of mode **out** because these may be known on entry, for example `A'FIRST` when `A` is an array. This simple formulation is valid for the current version of Penelope, however, since attributes are not yet supported.

The ⟨*out annotation*⟩ can be used to annotate both procedures and functions, although one must use a result annotation to be able to refer to the value returned by a function.

6.2.4 Result annotations

⟨*result annotation*⟩ ::=
  --| return ⟨*identifier*⟩ **such that** ⟨*assertion*⟩;

where the assertion is a two-state predicate. The only Ada variables allowed to appear in the assertion are those variables appearing in the formal parts of the subprogram declaration and the subprogram's side effect annotation, and ⟨*identifier*⟩. The ⟨*identifier*⟩ may not appear in the assertion modified by **in**, since it is senseless to talk about the value on entry of the thing returned.

The result annotation may annotate only functions. It is exactly like the out annotation except that ⟨*identifier*⟩ stands for the return value. In principle the result annotation renders the out annotation superfluous for functions, but the out annotation may be convenient for describing side effects of the function. If the ⟨*result annotation*⟩ is omitted, that is equivalent to a ⟨*result annotation*⟩ with an assertion of **true**.

The sort of ⟨*identifier*⟩ is the sort on which the return type of the function is based.

There is a short form result annotation

⟨*result annotation*⟩ ::= **--| return** ⟨*term*⟩;

which is equivalent to

⟨*result annotation*⟩ ::=
  --| return ⟨*identifier*⟩ **such that** ⟨*identifier*⟩ = ⟨*term*⟩;

where ⟨*identifier*⟩ is not free in ⟨*term*⟩. Thus the annotation

--| return $x**y$;

is equivalent to

--| return z **such that** $z = x**y$;

6.2.5 Propagation constraints

⟨*propagation constraint*⟩ ::=
  ⟨*constraint propagation annotation*⟩
  | ⟨*strong propagation annotation*⟩
  | ⟨*exact propagation annotation*⟩

These annotations specify under what conditions the subprogram may terminate by propagating an exception. The first supplies necessary, the second sufficient conditions for termination by exception. The third, the most commonly used, supplies necessary and sufficient conditions for termination by exception, assuming that the subprogram terminates at all.

6.2.6 Constraint propagation annotation

⟨*constraint propagation annotation*⟩ ::=
  --| **raise** $[[\langle exception\rangle]]^{+}_{|}$ => **in** ⟨*assertion*⟩;

where ⟨*assertion*⟩ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes **in** or **in out**.[2]

If the subprogram terminates by propagating any of the exceptions listed, the entry state must have satisfied the assertion. Verification conditions will be generated whose truth will guarantee that the subprogram cannot propagate any of the exceptions listed unless it is called in a state satisfying the assertion.

6.2.7 Strong propagation annotation

⟨*strong propagation annotation*⟩ ::=
  --| **in** ⟨*assertion*⟩ => **raise** $[[\langle exception\rangle]]^{+}_{|}$;

where ⟨*assertion*⟩ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes **in** or **in out**.

When the entry state satisfies the assertion, the subprogram *must* raise one of the exceptions listed, if it terminates. Therefore, strong propagation annotations for disjoint sets of exceptions cannot be satisfied unless they have mutually exclusive assertions. Verification conditions will be generated whose truth will guarantee this exclusivity, and will guarantee

[2] In the propagation annotations, **in** acts as a syntactic marker, not as a modifier.

that every time the subprogram is called in a state satisfying the assertion it will propagate one of the exceptions listed if it terminates at all.

6.2.8 Exact propagation annotations

$\langle$*exact propagation annotation*$\rangle$ ::=
  --| raise [[$\langle$*exception*$\rangle$]]$^{+}_{|}$ **<=> in** $\langle$*assertion*$\rangle$;

or

$\langle$*exact propagation annotation*$\rangle$ ::=
  --| in $\langle$*assertion*$\rangle$ **<=> raise** [[$\langle$*exception*$\rangle$]]$^{+}_{|}$;

where $\langle$*assertion*$\rangle$ is not a two-state predicate. All Ada variables in the assertion take their values from the entry state. The only Ada variables allowed to appear in the assertion are the global or formal parameters of modes **in** or **in out**.

This annotation is an abbreviation for the conjunction of the strong propagation annotation and the constraint propagation annotation with the same list of exceptions and the same assertion. The same interpretations and restrictions apply; the intent is that the assertion be a necessary and sufficient assertion for the propagation by the subprogram of one of the exceptions listed, if the program terminates.

6.2.9 Propagation promises

A *propagation promise* makes claims about a subprogram's exit state if it terminates by propagating an exception.

$\langle$*propagation promise*$\rangle$ ::=
  --| raise [[$\langle$*exception*$\rangle$]]$^{+}_{|}$ [[**=> promise** $\langle$*assertion*$\rangle$]];

where $\langle$*assertion*$\rangle$ is a two-state predicate. Unmodified variables take their values from the exit state. If the **promise** clause is omitted, that is equivalent to a **promise** clause with an assertion of **true**. (This asserts that the subprogram may propagate the exception but leaves the resulting state unspecified.) If the subprogram terminates by propagating any of the exceptions listed, it does so in a state satisfying the assertion.

The following kinds of Ada variables may appear in the assertion:

- global parameters

- formal parameters of mode **in**
- formal parameters of mode **in out**, but modified only by **in**

Penelope does not allow you to make assertions about the value of formal **out** or **in out** parameters upon exceptional termination, because the methods by which (and the order in which) parameters are passed are implementation-dependent. The rules of Ada say that we must not rely on a particular method ([1, Sec. 6.2]). (There is no difficulty with global parameters, since they are always "passed by name".) Verification conditions will be generated whose truth will guarantee that the exit state satisfies the assertion whenever the subprogram terminates by propagating any of the exceptions named.

6.2.10 Subprogram declaration and body annotations

An Ada subprogram has a body and may have a separate declaration. (If there is no separate declaration, the subprogram body is both declaration and body.) In this section we address the issue of consistency of annotation (specification) for the subprogram declaration and body.

In Penelope the annotation of the subprogram's declaration is the *external* specification of the subprogram. That is, calls on the subprogram must satisfy the entry conditions specified in that subprogram annotation and may assume the exit conditions specified. The annotation of the subprogram body is the *internal* specification: verification conditions assure that the implementation of the subprogram satisfies this specification. What remains to be assured is that the specification of the body is at least as strong as that of the declaration. There are three cases:

- There is no separate subprogram declaration, hence the subprogram has just one subprogram annotation.
- We do not provide a subprogram annotation for the subprogram body, which inherits the annotation of the subprogram declaration. (Penelope notes this by displaying three asterisks at the annotation of the subprogram body.)
- We provide the subprogram body with an annotation that differs from that of the subprogram declaration (it may be easier to prove a stronger specification). In this case we must ensure that the specification of the subprogram body is at least as strong as that of the declaration.

In order to show that the annotations of the body imply those of the declaration we must show that

1. the in conditions for the declaration implies the in condition for the body;

2. for each way τ in which the subprogram may terminate (where τ represents either normal termination or termination by raising some particular exception) the in conditions for the declaration and the body's exit conditions for τ must imply the declaration's exit conditions for τ.

There is a slight subtlety in defining what we mean by the exit conditions associated with each way of terminating the program. For example, the annotation **raise** E <=> **in** P not only defines an exit condition associated with termination by propagating E, but also adds $\neg$ **in** P to the exit conditions associated with all other ways, normal or exceptional, of terminating the subprogram.

6.3 Internal annotations

6.3.1 Loop invariants

Each **loop** statement in an Ada program requires an *invariant*, similar to the invariants of [2] and [6]. Intuitively, the invariant states what must be true before every evaluation of the iteration scheme and execution of the loop body.

You provide an invariant using the **invariant** keyword:

--| **invariant** ⟨*assertion*⟩;

By default the loop invariant is a "cut-point." That is, the invariant becomes the precondition of the loop, and Penelope generates a separate verification condition whose proof justifies that precondition. This verification condition says that

1. the invariant is preserved by the evaluation of the iteration scheme (if any) and the loop body, and

2. if the loop terminates, the postcondition of the loop holds.

This correctly handles the possibility that the loop is exited by means of a return statement, an exit statement, or an exception and the possibility that evaluation of the iteration scheme raises an exception or has a side-effect.

When the invariant is a cut-point, the loop becomes one of the modules of the program proof and the loop invariant serves as stand-alone documentation of the loop's effect. The only drawback is that this sometimes requires packing rather a lot of information into the invariant. Experimentally, therefore, we offer the opportunity to "lump" the verification condition for the loop. The invariant is no longer a cut-point, and the loop is no longer a separate module of the proof. Instead, the precondition of the loop becomes not the invariant but the loop verification condition itself (which is no longer required to have a stand-alone

proof). The advantage is that a weaker invariant will often suffice for a loop that has been lumped. To lump a loop, click on `Lump` on the help-pane menu. To switch back from a lumped loop to an "unlumped" one, click on `UnLump` on the help-pane menu. Note that lumping a loop causes the proof of the loop verification condition to be lost.

6.3.2 Sending information forward

Predicate transformation works backward, from a description of a goal to a description of the preconditions sufficient to achieve the goal. At times that is frustrating. If, for example, a subprogram has $\mathtt{x} > 0$ as one of its in conditions, you might like to apply that fact to simplify the preconditions Penelope generates. The in condition will, of course, be applied eventually, but until it is applied the preconditions are more complicated than necessary, and reasoning about the program more difficult. We therefore provide three simple ways in which information can be "sent forward."

6.3.2.1 Embedded assertions

We may strengthen the claims made in a subprogram annotation by using an *embedded assertion*. Syntactically, an embedded assertion is a formal comment, thus:

$$\langle \textit{embedded assertion} \rangle ::= \quad \texttt{--|} \ \langle \textit{assertion} \rangle \, ;$$

The embedded assertion may appear only in the position of a statement in a sequence of statements. It asserts claims that the ⟨*assertion*⟩ is true whenever control reaches that point in the program.

Formally, the effect of an embedded assertion is to conjoin the ⟨*assertion*⟩ to the precondition, which guarantees that you will indeed be required to show that the ⟨*assertion*⟩ is true whenever control reaches the point of the embedded assertion. This strengthening of the precondition can sometimes be used to make the precondition much simpler (e.g., by ruling out an impossible branch of a conditional term).

The embedded assertion is a two-state predicate (see Section 5.5).

6.3.2.2 Cut-point assertions

A *cut-point assertion* is, unsurprisingly, a cut-point. That is, the effect of a cut-point assertion is to *replace* the precondition with the ⟨*assertion*⟩. The cut-point assertion says, like an embedded assertion, that the ⟨*assertion*⟩ is true whenever control reaches it. In addition, it says that this is all one needs to know–the truth of the ⟨*assertion*⟩ suffices to guarantee that the desired exit conditions will hold if the program terminates after control has reached this point.

Syntactically, a cut-point assertion is a formal comment, thus:

$\langle$*cut point assertion*$\rangle$ ::= **--| assert** $\langle$*assertion*$\rangle$;

Cut-point assertions may appear where embedded assertions may appear. A separate verification condition is generated for each cut-point assertion; the user must show that the $\langle$*assertion*$\rangle$ implies the precondition that it has replaced.

6.3.2.3 Local lemmas

A *local lemma* is an invariant that holds at each control point in a certain scope–namely, from the point at which it is declared to the end of the sequence of statements in which it is declared.[3] Consider the illustration with which this section began: If `x` $>$ 0 is an in condition of a subprogram, then the statement `IN x` $>$ 0 can become a local lemma true over the entire sequence of statements of the subprogram.

The syntax of a local lemma is

$\langle$*statement*$\rangle$ ::=
--| lemma $\langle$*identifier*$\rangle$ [[**rewrite**]]: $\langle$*term*$\rangle$;

The identifier provides a name by which the lemma may be invoked in proofs undertaken within its scope.

We guarantee that the truth value of the $\langle$*term*$\rangle$ is invariant by a syntactic check: the term may not contain occurrences of any variable that is potentially modified within the scope of the local lemma–intuitively, we check it as though it were an **in** parameter of its scope. We guarantee that it is invariantly *true* by conjoining the $\langle$*term*$\rangle$ to the current precondition.

The optional rewrite indication states that the local lemma should be added to the current set of rewrite rules.

6.4 Annotations of packages

Currently the only annotations of packages available are annotations of private types.

6.4.1 Annotations of private types

Ada private types require special treatment because of their dual nature. If package `p` declares a private type `t`, then in the visible part of `p`, and in clients of `p`, only the equality

[3]Currently, a local lemma may be declared only within a sequence of statements.

operation can be used on objects of type `t`. The structure of such objects is hidden.[4] We will call this view of `t` the *abstract* view. Within the body of `p`, however, type `t` is treated as an ordinary Ada type, declared in the private part. We call the part of the package that is not visible (the body and the private part of the declaration) the *hidden* part and this view of `t` the *concrete* or *representation* view.

We wish to specify the subprograms of package `P` using the abstract theory of `t`. We must, however, *implement* those subprograms by using operations on the representation view of `t`. Furthermore, we have to *prove the implementation* by reasoning about those operations, which requires a concrete theory of `t`. Therefore we use two different sorts to model the private type `t`.

We define the *abstract sort* by annotating the declaration of `t` with the name of the sort on which it is *based*. When declaring a private type we write

⟨*declarative item*⟩::=
  type ⟨*identifier*⟩ **is private** ;
  --| **based on** ⟨*sortmark*⟩;

The *concrete sort* is defined by the declaration of `t` in the private part, in the usual way.

We describe the connection between the concrete sort, used to model the behavior of the type within the hidden part of the package, and the abstract sort by specifying an *abstraction function* and a *representation invariant*, as described by Hoare [14]. For every private type, Penelope supplies templates for the abstraction function and representation invariant for that type. The abstraction function defines an abstract value for each member of the concrete type `t`. The representation invariant takes an argument of the concrete sort and returns **true** if the argument can represent an abstract value, **false** otherwise.

In computing preconditions and verification conditions in the visible part of the package and in clients of a package, Penelope treats objects of private type as objects of the abstract sort. This treatment is possible even though Ada semantics makes the representation visible, albeit obliquely, outside of the package, because Penelope defines restrictions on references to objects of private type sufficient to ensure that they can be treated, in the visible part of the package and in the package clients, as objects of the abstract sort.

In the body of the package, the objects of the private type are treated as objects of the concrete sort. Penelope automatically translates annotations from the visible part of the package, using the abstraction and invariant functions provided.[5]

[4] This is an oversimplification. Equality on objects of private types is defined by identity of representation, so the internal structure of such objects is not as hidden as one might expect.

[5] The automatic translation does not extend to compound types whose components are private types. If you define a private type, say `foo`, and then declare a type of arrays of `foo`, the translation does not extend properly to elements of the array type.

How this translation works is best explained with an example. Consider a package that implements a stack type. Assume that a definition of type `intarray`, an integer-indexed array of integers, is available. Then,

```
package stacks is
   type stack is private;
    --| based on Stack;
private
   type stack is record
    depth :  integer;
    contents:  intarray;
  end record;
    --| abstraction function : stack_abs;
    --| representation invariant : stack_inv;
end stacks;
```

This says that type `stack` is based on the abstract sort *Stack*. Let's call its concrete sort *CSort*. Then our specification says that there are two mathematical functions, *stack_abs* and *stack_inv*. The first one has the signature $CSort \rightarrow Stack$. Its definition, found in the mathematical part of our specification, defines our choice of representation for stacks. Section A.2 gives an abstraction function and representation invariant for an implementation of stacks as records.

The function *stack_inv* has the signature $CSort \rightarrow Bool$. Given any object of type `stack`, this function will return true if and only if that object can represent a stack. A reasonable definition might be $stack\_inv(r) = (r.depth >= 0)$.

Now suppose that we have a mathematical theory of stacks available (see Figure A.1) that provides the function *top*, which returns the highest element on a stack, and the function *pop*, which returns the rest of the stack. We can then define the effect of an Ada procedure pop:[6]

```
procedure pop(n :  out integer; s :  in out stack);
  --| where
  --|    out s = pop(in s);
  --|    out n = top(in s);
  --|    raise stack_empty <=> in s=empty();
  --| end where;
```

In the body of the package, the annotation is automatically translated to the following concrete form.[7] We require that s represent a stack on entry to pop, and promise that on exit it will continue to represent a stack. The effect of pop is defined in terms of the mathematical functions, and reasoning about the program is carried out in terms of those functions and the representation functions *stack_abs* and *stack_inv*.

```
procedure pop(n :  out integer; s :  in out stack);
  --| where
  --|    in stack_inv(s);
  --|    out stack_inv(s);
  --|    out stack_abs(s) = pop(stack_abs(in s));
  --|    out n = top(stack_abs(in s));
  --|    raise stack_empty <=> in (stack_abs(s)=empty());
  --| end where;
```

6.5 Annotations of compilation and library units

Ada programs are modular, and we would like to verify them in a modular way. In Penelope the Ada term *compilation unit* is generalized to include units of mathematics. Ada compilation units are verified individually and stored, with the mathematical units, in an external library similar to the Ada library. When we verify an Ada program, we verify each of its compilation units with respect to previously verified units, and then we verify the composition of the compilation units. In particular, we have to be concerned with the effect of

[6]Note that there can be no confusion between the Ada procedure pop and the mathematical function *pop*: in Ada code the procedure is always denoted; in the annotation the mathematical function is always denoted.

[7]The abstract form of the annotation will appear in the package body, but the concrete form is used internally and will appear in preconditions and verification conditions.

elaborating the library units prior to the execution of a main program, since the elaboration itself may have arbitrarily complex effects.

Each Penelope buffer is associated with an external Penelope library, and each point in a Penelope buffer is associated with a *current library*, which is determined by the buffer's external library and by the compilation units occurring before that point in the buffer. (The precise rules are explained below.)

If a Penelope session simply adds new units to a semantically consistent external library, the semantic consistency of the resulting library will be maintained. If, however, a Penelope session overwrites a library unit already residing in an external library, the consistency of the resulting library is no longer guaranteed, but must be reestablished (if possible) by running a script that rebuilds the library by replaying the changes through Penelope in the proper order.

A Penelope library must not have two traits with the same simple name or two Ada library units with the same simple name (but may have a trait and an Ada unit with the same name). This invariant will be maintained in all external libraries that are modified only by using Penelope's `write-library` command, and it is true of the current library at any point so long as it is true of the external library.

6.5.1 Library annotation

Verification, like compilation, takes place with respect to a library. An annotation at the head of each file used by Penelope provides the Unix pathname of the library directory. The library annotation identifies the library used for verification.

⟨*library annotation*⟩ ::=
  --| **library** ⟨*Unix pathname*⟩;

If no library annotation is present, an empty external library is assumed.

6.5.2 The current library

The Penelope buffer contains a candidate change to the external library. At any point in the buffer, the *current* candidate change consists of all the candidate definitions of library units in the buffer up to that point, and if the same unit has more than one candidate definition in the buffer, the *first* such definition is the valid one. (Penelope displays a warning message when the user tries to define a compilation unit that is going to be ignored.) The current library at any point in the buffer is the library that would result from updating the external library by the current candidate change.

Here is a more operational definition, describing how to determine, at any point in the Penelope buffer, what the name of a compilation unit denotes: If `foo` is the name of a unit in the buffer (a unit of the proper kind, either trait or Ada unit), then it denotes the *first* such unit. Otherwise `foo` denotes the unit of the proper kind named `foo` in the buffer's associated external library, if any.

The effect of the `write-library` command is actually to update the library by the candidate change current at the end of the buffer.

6.5.3 Context clause annotations

An Ada compilation unit includes a context clause stating which other Ada compilation units are needed for its compilation. Analogously, in specifying a compilation unit, we can state what specification information from the Penelope library is required to state the specification. The *context clause annotation* enables us to refer to mathematics defined in *traits* of the Larch Shared Language (see Chapter 7) and to the specifications of Ada compilation units.

⟨*context clause annotation*⟩::=
  --| **with** [[*theory reference*]]$^{+}_{,}$ [[([[⟨*rename*⟩]]$^{+}_{,}$)]]
⟨*theory reference*⟩::=
  trait ⟨*identifier*⟩
  | **spec** ⟨*identifier*⟩

The first kind of theory reference imports a trait needed for the specification, for example

--| with trait *Stack(Int* **for** *Elem);*

Here *Stack* must be a trait in the current Penelope library, and the renaming *(Int* **for** *Elem)* substitutes the symbol *Int* for *Elem* throughout, as explained in Section 7.2. There are two reasons for such renamings: One is, essentially, parameterization—we write a general trait describing stacks of arbitrary elements and then specialize it to the case in which the elements are integers. The other is to avoid name conflicts with symbols introduced by other included traits.

The second kind of theory reference imports a specification. In the current version of Penelope there is only one use for this annotation—namely, the need to resolve name conflicts by relabeling the specification of a compilation unit. Suppose, for example, that package `A` "withs" packages `foo` and `bar`, whose specifications are each written in terms of an operation called *top*. Suppose, further, that the theories of `foo` and `bar` assign different, possibly contradictory, meanings to `top`. We can relabel the specification of `foo` by saying

--| with spec *foo (foo_top* **for** *top);*

The effect of this is that `A` sees a new, but equivalent, specification of `foo`, in which *foo_top* replaces *top* both in the theory and in the Larch/Ada annotations.

In a "**with spec**" annotation the renaming of sorts and functions, if any, must constitute a signature isomorphism. That is, there must be a one-to-one mapping between names in the old theory and names in the new (renamed) theory. This guarantees that the specification of `foo` visible to `A` is really equivalent to the specification of `foo` in the library.

Names of Ada library units are case insensitive. Names of traits are case sensitive.

6.5.4 The theory of a compilation unit

Every compilation unit has an associated mathematical theory that defines the language of the annotations, including axioms and lemmas that are available for proofs. When we speak of "the theory of a unit," we mean this mathematical theory. The theory of an Ada compilation unit `U` is the union of the following theories:

1. an initial theory for Ada (including, for example, sort *Int*),
2. the theories of the library units mentioned in the Ada context clause for `U`—except for any unit `foo` that is also mentioned in an context clause annotation of the form "**with spec** `foo`",
3. if `U` is a body, the theory of the declaration for `U`, and
4. the theories of the units named in the context clause annotation (appropriately relabeled).

The effect of (2) and (4) together is that if `U` is both "**with** `foo`" and "**with spec** `foo` (f **for** g)", then the theory of `foo` is not included in the theory of `U`, but the theory of `foo` relabeled with f for g is included.

6.5.5 Main program annotation

In general, the meaning of a library unit depends on the whole program in which it runs. In particular, especially for package library units, it depends on all other compilation units of the program and the order in which they are elaborated. For example, if a unit is with package `P` and contains the declaration

```
x : t := p.y;
```

the value of `p.y` may depend on the effect of all the elaborations occurring before this declaration. Penelope therefore postpones considering the effects of elaborating a library unit

until the unit is to be elaborated in a main program. We verify the bodies of subprograms declared in library units, but postpone verification of elaboration code until a main program is identified.

The *main program annotation* causes Penelope to compute the effect of elaborating the library units of a subprogram if that subprogram were to be used as a main program. A main program annotation may appear wherever a compilation unit may appear.

$\langle$*main program annotation*$\rangle$ ::=
--| **main** $\langle$*identifier*$\rangle$
--| **where**
[[$\langle$*in annotation*$\rangle$]]*
--| **end where**;

The execution of a main program begins by elaborating all the library units it needs (if any) and then elaborating the main program itself. The in annotations of a main program annotation state what is assumed to be true before any of this elaboration takes place. Since no program variables are visible in this state, the in annotations can only discuss the state of accessible external entities like the file system. The effect of elaborating the library units is computed at the point of the main program annotation and a verification condition is generated, stating that after elaboration of library units mentioned in the context clause of the main program, the entry condition of the main program holds.[8]

To declare a main program, click on `main-program` on the help-pane menu. When you fill in the name of the main program, a message appears stating that the elaboration order needs to be recomputed. Click on the message, then click on `compute-elaboration-order` on the help-pane menu. This command causes Penelope to compute a valid elaboration order and display a sequence of lines of the form

--! elaborate $\langle$*compilation unit kind*$\rangle$ $\langle$*identifier*$\rangle$

You can inspect and simplify the preconditions of elaborating the library units just as you would preconditions of executable statements. This can be useful if the verification condition for the main program is difficult or impossible to prove.

6.6 Annotations of generic units

Penelope supports generic declarations and instantiations occurring as compilation units.

[8] In the future it will be possible to allow the effect of elaboration to remain implicit, which is more convenient when, for example, a large number of objects are initialized during elaboration.

6.6.1 Generic declaration

⟨*generic_specification*⟩ ::=
  ⟨*generic_formal_part*⟩
  ⟨*subprogram_declaration*⟩
| ⟨*generic_formal_part*⟩
  ⟨*package_declaration*⟩
⟨*generic_formal_part*⟩ ::=
  [[⟨*formal_trait*⟩]]
  [[⟨*generic_parameter_decl*⟩]]*
⟨*formal_trait*⟩ ::=
  --| formal trait
  ⟨*trait_body*⟩
  --| end formal trait
⟨*generic_parameter_decl*⟩ ::=
  ⟨*idlist*⟩ : ⟨*mode*⟩ ⟨*typemark*⟩
  | **type** ⟨*identifier*⟩ **is** ⟨*generic_type_definition*⟩
  | **--| lemma** ⟨*labelled_term*⟩

Generic declarations admit two kinds of annotations:

- Annotations of the generic formal parameters state restrictions on the values of the actuals with which they may be instantiated.
- The generic declaration's subprogram or package specification is annotated like an ordinary subprogram or package specification. It serves as a template for the annotations of generic instances.

The current implementation of Penelope permits only certain kinds of generic formal parameters:

- object parameters of mode **in**;
- formal private types
- formal discrete types
- formal integer types
- array types constructed from non-generic types and the above formal types

In Ada the declarations of generic formal parameters stipulate certain kinds of restrictions on the actuals, but these restrictions are not very expressive. For example, we may require

that a generic object parameter be instantiated by actuals lying within a certain integer subtype, but we have no way to require that the actual be a prime number, although such a restriction may be essential for the correct functioning of the generic instance.

Larch/Ada annotations permit us to formulate more expressive restrictions on actual parameters. These restrictions may be thought of as in conditions on the **is new** operator that is used to elaborate instances of the generic. Syntactically, these annotations take the form of local lemmas occurring within the sequence of generic parameter declarations. Semantically, they place the following requirement on any generic instantiation: the assertions must be true in the state occurring immediately after all the actual parameters to the instantiation have been evaluated, and before elaboration of the generic instance itself.

The annotations are labelled, like local lemmas (see Section 6.3.2.3), so that you can appeal to them during proof of the generic body.

A generic declaration may also contain a formal trait whose primary role is to specify generic formal subprogram parameters, which are not yet implemented.

6.6.2 The body of a generic

Annotating and proving the correctness of the body of a generic are just like annotating and proving the correctness of the bodies of ordinary units. When proving the body, the local lemmas stating requirements on the formal parameters are part of the available theory.

Certain logical checks that formally express the difference between generic units and ordinary units are not implemented.

6.6.3 Generic instantiation

⟨*generic_instantiation*⟩ ::=
  ⟨*instantiation_kind*⟩ ⟨*identifier*⟩ ⟨*name*⟩ [[⟨*generic_actual_part*⟩]]
  [[⟨*fitting*⟩]]
⟨*instantiation_kind*⟩ ::= **function** | bf procedure | **package**
⟨*generic_actual_part*⟩ ::=
  ($[[\langle expression \rangle]]^{+}_{,}$)

Conceptually, the elaboration of a generic instantiation proceeds as follows: The body of the generic is used as a template to create the body of the instance by appropriately substituting actual for formal parameters; the actual parameters are checked against the constraints defined by the definitions of the formals; the body of the instance is elaborated. Annotating and proving the correctness of a generic instantiation involve analogous logical operations.

First, the user may supply a *fitting* that renames sorts and operations occurring in the

annotations of the generic. This is analogous to supplying actual parameters for the generic formals, because the annotation of the generic is only a template for the annotation of its instances. As far as Penelope is concerned, the semantic meaning of the instantiation is the annotation that results from applying this relabeling to the annotation of the generic's subprogram or package.

Second, the creator of a generic instantiation must show that the generic actual parameters satisfy all the restrictions stated in the annotations of the generic formals. As noted above, these restrictions are essentially preconditions of the elaboration of the generic instance. A verification condition is generated to ensure that the restrictions are met.

Chapter 7

The Larch Shared Language

To complete the description of Penelope specifications we must describe the syntax and semantics of the traits in which the necessary specification mathematics is defined and developed. In Penelope traits are written in a variant of the Larch Shared Language (LSL).

A complete introduction to LSL is beyond the scope of this manual; [11] provides such an introduction. In this manual references to LSL or the Larch Shared Language refer to the variant implemented in Penelope. This manual gives a very brief introduction to (or reminder of) the meaning of each syntactic construct. More complete explanations are given for constructs that Penelope has added to LSL in support of Ada verification.

The proof obligations necessary for mathematically sound proofs of the correctness of Ada programs are documented in [9]. Penelope does not insist on mechanical proof of, or even document, some of those obligations—for example, the obligation to show that certain traits are logically consistent.

7.1 Traits

In LSL, mathematics is developed in modules called *traits.* A trait denotes a mathematical theory. A trait typically introduces a new sort and/or new functions on a sort, with their defining axioms and lemmas that will be useful in verifying programs specified using the functions.

A trait appears in Penelope in the place of an Ada compilation unit.

$$\langle compilation\_unit \rangle ::= \quad \langle trait \rangle$$

Traits and Ada compilation units may be interspersed in the Penelope buffer. As with Ada compilation units, traits are entered into libraries, from which they may be referenced by later

traits and compilation units through context clauses. "Later" in this case means appearing later in a sequence of compilation units, or referenced through a library (see Chapter 2).

$\langle\textit{trait}\rangle$ ::=
  --| Larch
  $\langle\textit{identifier}\rangle$ $[[\langle\textit{trait\_parameter\_part}\rangle]]$: **trait**
    $\langle\textit{trait body}\rangle$
$\langle\textit{trait\_parameter\_part}\rangle$ ::= $(\ [[\langle\textit{identifier or function name}\rangle]]^{+}_{,}\ )$
$\langle\textit{trait body}\rangle$ ::=
  $[[\langle\textit{trait\_context}\rangle]]^{*}$
  $[[\langle\textit{function\_def}\rangle]]^{*}_{;}$
  $[[\langle\textit{prop\_part}\rangle]]$
  $[[\langle\textit{consequences}\rangle]]$
  $[[\langle\textit{proof\_section}\rangle]]$
  --| end Larch

Comments may appear immediately following the keyword **trait**.

Figure 7.1 shows a trait for lists.

7.2 Building on previous traits (includes and assumes)

Traits typically refer to sorts and/or theorems defined in other traits. For example, in Appendix A the trait *Stacks* uses the definition of lists in trait *Lists*.

When trait B includes trait A, the axioms of A become axioms of B. Assumptions of A (that is, axioms of traits *assumed* by A) become unproved lemmas of B and lemmas of A (proved or unproved) become proved lemmas of B.

When trait B assumes trait A, then axioms and assumptions of A become assumptions of B and lemmas of A become lemmas of B.

$\langle\textit{trait\_context}\rangle$ ::=
  includes $[[\langle\textit{trait\_ref}\rangle]]^{+}_{,}$
  | **assumes** $[[\langle\textit{trait\_ref}\rangle]]^{+}_{,}$
$\langle\textit{trait\_ref}\rangle$ ::=
    $([[\langle\textit{traitname}\rangle]]^{+}_{,}\ )\ [[\ ([[\langle\textit{rename}\rangle]]^{+}_{,}\ )]]$
$\langle\textit{traitname}\rangle$ ::=
    $\langle\textit{identifier}\rangle$

Each trait reference must be to a trait that has been previously defined.[1] Note that traits underlying the semantics of Penelope's Ada subset are built in and automatically included

[1] That is, the trait must be found in a previous "compilation unit" in Penelope's buffer, or in the library.

```
--| Larch
Lists: trait
  introduces
    nil :    -> List
    cons :    Elem, List -> List
    tail :   List -> List
    head :    List -> Elem
    length :    List -> Elem
    append :     List, List -> List
  asserts
    List freely generated by nil, cons
    forall l, l1, l2:List, e:Elem
      head (rewrite): head(cons(e,l)) = e
      tail (rewrite): tail(cons(e,l)) = l
      length0: length(nil) = 0
      length: length(cons(e,l)) = length(l) + 1
      append0 (rewrite): append(nil(),l) = l
      append: append(cons(e,l1),l2) = cons(e, append(l1,l2))
  implies
    forall l:List, e:Elem
      length_non_neg: length(l) >= 0
      append_to_nil: append(l, nil()) = l
--| end Larch
```

Figure 7.1: A trait for lists

in every trait, and may not be mentioned in trait references. A trait must not include or assume itself circularly, either directly or indirectly through other traits.

Penelope will compute the proof obligations incurred when you include a trait that contains assumptions. See Section 7.8.

7.2.1 Renaming sorts and function names

When we include or assume a trait, we often want to rename some of its sorts or function names. Using trait *Lists*, for example, we can build a theory for a list of integers by including *Lists*, renaming *Elem* to *Int*.

> **includes** *(Lists) (Int* **for** *Elem)*

Function names can be overloaded, so renaming functions is only necessary when two distinct functions have the same name *and* the same signature. We may also rename for convenience. For example, the *Stacks* trait in Appendix A renames *cons* to *push*.

In general we have:

$\langle rename \rangle$ ::=
  $\langle sortmark \rangle$ **for** $\langle identifier \rangle$
| $\langle identifier \rangle$ **for** $\langle function\ name \rangle$

Typically renaming a sort or function name simply substitutes one identifier for another. We may, however, sometimes need to provide a sortmark that is not an identifier (a rather contrived example would be a list of enumeration literals), or we may need to provide a full name (name with signature) for the function we are replacing. Note that it is not possible to rename the predefined sorts like *Int*.

7.2.2 Renaming traits

$\langle traitname \rangle$ ::=
  $\{\langle identifier \rangle\}\langle identifier \rangle$
| $\langle identifier \rangle$ **is new** $\langle identifier \rangle$

The lemmas and axioms (for short, the *theorems*) in a Penelope trait have names so that we can reference them in proofs. If a theorem named `foo` occurs in a trait called `bar`, its full name is "*foo* in trait *bar*." If two theorems have the same name, one of them will be unavailable when doing proofs, so we need a mechanism for renaming theorems.

Suppose, for example, that we are interested in lists of integers and also lists of booleans. We could write

T : **trait**
  includes *(List) (Int* **for** *Elem, IntList* **for** *List)*
  includes *(List) (Bool* **for** *Elem, BoolList* **for** *List)*

Trait *List* contains an axiom *head* saying that for every *e:Elem* and *s:List*, $head(cons(e, s)) = e$. Neither inclusion nor the renamings of sorts and operations renames theorems. Therefore, the effect of the inclusions and renamings in trait T is that T contains two theorems whose full name is "*head* in trait *List*"—one stating the principle for integer lists and one for boolean lists.

We enable the user to rename lemmas by saying

includes *(IntList* **is new** *List)(Int* **for** *Elem, IntList* **for** *List))*

The effect of "*IntList* **is new** *List*" is to replace *List* wherever it occurs in the name of a lemma. That is, it changes every lemma name "*foo* of trait *List*" into the name "*foo* of trait *IntList*." It is important to notice that this renaming does *not* affect the names of any other theorems of *List*. If, as a result of including trait S, *List* contains a theorem named "*foo* of trait S" this theorem is not renamed.

When the cursor is positioned at a ⟨*traitname*⟩ placeholder, the menu item `rename-trait` appears. Clicking on `rename-trait` provides a template for trait renaming with **is new**. When the cursor is positioned at a ⟨*rename*⟩ placeholder, the menu item `trait-renaming` appears. Clicking on this menu item provides a template for trait renaming.

There are times when one wants a convenient way to rename every theorem of a trait. The following experimental notation is available:

includes *({Int}List)(Int* **for** *Elem, Int* **for** *List)*

The effect of this is to include trait *List*, relabeling *all* its theorems: "*foo* of trait S" becomes "*foo* of trait *IntS*." That is, the sequence of characters within the braces is prefixed to the trait part of each theorem name. When the cursor is positioned at a ⟨*traitname*⟩ placeholder, the menu item `prefix-trait` appears. Clicking on `prefix-trait` provides a template for adding a prefix, enclosed in braces, to the trait name.

7.3 Sort declarations

Most sorts do not have to be declared. In the *List* example above, the sort *List* is never explicitly declared. *Sort declarations* are a Penelope extension to LSL. Sort declarations enable us to define *synonyms* or *nicknames* for sorts. Built-in names of sorts may be too long to be practical; for instance the name of the predefined sort corresponding to the Ada type `character` is a comma-separated list of 128 ASCII characters. For this sort we use the *nickname Char*.

⟨*trait_context*⟩ ::= ⟨*sort_declaration*⟩
⟨*sort_declaration*⟩ ::=
    sort ⟨*identifier*⟩ **is** ⟨*sortmark*⟩ ;

LSL offers a *shorthand* facility for defining enumeration and tuple (record) sorts. Sort declarations may be viewed as a combination of LSL's shorthand facility with a general capability for providing synonyms for sort names.

7.4 Function declarations (introduces)

In order to define a function, we need to declare its *signature* and provide its meaning. A function declaration gives the signature of a function. Its meaning is supplied by the axioms that follow.

⟨*function declarations*⟩ ::=
  introduces
  [[[[⟨*applicator*⟩]]$^{+}_{,}$: ⟨*signature*⟩;]]$^{+}$
⟨*applicator*⟩ ::=
  ⟨*identifier*⟩
 | ' ' ⟨*operator symbol*⟩ ' '

A function is uniquely identified by the combination of its applicator and signature. A function may be declared more than once in different traits. Renaming may be necessary to avoid name clashes between functions when different traits are combined.

Note that several functions can be declared in a single declaration.

7.5 Signatures

The *signature* of a function gives the sorts of its arguments and result. For example, consider the function *is_prime*, which, given an integer, returns **true** or **false**, depending on whether the integer is prime. It has the signature $is\_prime : Int \rightarrow Bool$.

⟨*signature*⟩ ::=
  [[⟨*sortmark*⟩]]$^*_,$ -> ⟨*sortmark*⟩

Note that a *constant* is a function from no arguments to some domain. Zero, for example, has the signature ->*Int*.

7.6 Proposition part—Axioms

The propositions of a trait are its axioms. More precisely, in LSL we have both individual axioms and axiom schemes. The latter are used to provide the bases for proofs by induction and extensionality. Mathematically, the axioms of a trait must be consistent. This is the responsibility of the user. Penelope does not check the consistency of axioms.

The axioms may be schemes (see below), variable-free equations, or universally quantified equations.

⟨*prop_part*⟩ ::=
  asserts
    [[⟨*scheme*⟩]]*
    [[⟨*eq_part*⟩]]
⟨*eq_part*⟩ ::=
    [[⟨*vbl_free_equations*⟩]]
    [[⟨*quantified_eq_seqs*⟩]]
⟨*vbl_free_equations*⟩ ::=
  equations:
    [[⟨*labelled_theorem*⟩]]$^+$
⟨*quantified_eq_seqs*⟩ ::=
  [[**forall** ⟨*varlist*⟩]]
    [[⟨*labelled_theorem*⟩]]$^+$

7.6.1 Named axioms

Penelope extends LSL by giving each axiom a name (an identifier followed by a colon), by which it can be referred to in proofs. All names for axioms and lemmas (see Section 7.7) in a trait must be distinct.

⟨*labelled_theorem*⟩ ::=
    [[⟨*comment*⟩]]
    ⟨*identifier*⟩ [[(**rewrite**)]]:⟨*term*⟩

The ⟨*term*⟩ of each axiom is an assertion. Each group of axioms is preceded by a list of variables, which declares the variables that may appear free in the axioms. As indicated by

the syntax used, each axiom is assumed to be quantified over all the variables from that list that appear free in it. The variables are thus *bound* by the quantification. Each axiom may be commented.

7.6.2 Induction schemes—generated by

In our LSL the syntax for a **generated by** clause is as follows:

$\langle \textit{scheme} \rangle ::=$
$\quad \langle \textit{identifier} \rangle$ [[**freely**]] **generated by** $[[\langle \textit{function name} \rangle]]^{+}_{,}$

The first identifier names a sort.

For example, to say that all sets are generated by starting with the empty set and inserting successive elements into it we say

Set **generated by** *empty, insert*

A **generated by** scheme makes available proof by structural induction. To prove that all sets satisfy property P it suffices to show that the empty set has P and that whenever a set s has P, so does the set that results from inserting an element into s.

Each function name in a **generated by** clause must unambiguously name a function. The range of each function must be the generated sort. At least one function must have a domain in which the generated sort does not occur (e.g., a constant).

There can be more than one **generated by** scheme for a sort, in which case schemes are numbered starting with 1. Note that there may be **generated by** schemes for the same sort in more than one trait.

If we say

List **freely generated by** *nil, cons*

then we say not only that all lists are generated by pushing elements onto the nil list, but that two lists are equal only if they are generated by pushing the same elements onto the nil list in exactly the same sequence. Note that sets are *not* freely generated by *empty* and *insert*—since, e.g., we can generate the set $\{1, 2\}$ by inserting its elements in either order. The **freely generated by** scheme is an extension to LSL.

7.6.3 Well-founded relations

A *well-founded relation* is any binary relation R on some sort S, such that every non-empty subset of S has an R-minimal element (a least element with respect to R). This is equivalent to saying that there are no infinite chains $s_1, s_2, s_3, \ldots$, such that $\forall i :: R(succ(s_i), s_i)$. For example,

- $<$ is *not* a well-founded relation on *Int*, since $\ldots - 3 < -2 < -1$
- The relation between x and y defined by $\mathbf{abs}(x) < \mathbf{abs}(y)$ is a well-founded relation on *Int*.
- The relation between x and y defined by $\mathbf{abs}(x) \leq \mathbf{abs}(y)$ is not well-founded, since $\ldots \leq x \leq x \leq x$.
- The following two relations on lists are well founded: x is a shorter list than y; x is a proper initial segment of y.

Well-founded relations are interesting because they allow us to use general induction in proofs. In Penelope we can state, either as an axiom or as a lemma, that a relation is well-founded. Then induction over elements of the sort can be based on that relation.

$\langle axiom \rangle$ ::=
  well-founded $\langle function\ name \rangle$;

An error message is generated if the function name does not represent a relation, that is, if it is not a binary function over some sort, with range *Bool*.

7.6.4 Partitioning schemes—partitioned by

The **partitioned by** assertion

$\langle scheme \rangle$ ::=
  $\langle identifier \rangle$ **partitioned by** $[[\langle function\ name \rangle]]^{+}_{,}$

says that the "observer functions" given in the list of function names are sufficient as a group to distinguish between elements of the sort named by the identifier. For example, the following two assertions are true (given our ordinary understanding of lists and sets):

Set **partitioned by** *member*
List **partitioned by** *is_empty, head, tail*

The first says that two sets having the same members are equal. The second is true because two empty lists are equal; and two (nonempty) lists with the same head element and the same tail are equal.

There can be more than one **partitioned by** scheme for a sort, in which case schemes are numbered starting with 1. Each scheme is independent of the others; that is, each set of functions is sufficient to distinguish to distinguish between elements of the sort. Note that there may be partitioning schemes for the same sort in more than one trait.

Each function name in a **partitioned by** clause must unambiguously name a function. The domain of each function must include the partitioned sort. At least one function must have a range that is not the partitioned sort.

In Penelope proofs, the **partitioned by** scheme is used in proofs by extensionality.

7.6.5 Continuity

We can claim as an axiom that a function is continuous.

⟨*scheme*⟩ ::=
  continuous ⟨*function_name*⟩

A ⟨*function_name*⟩ consists of an identifier and an optional signature. If the identifier is ambiguous a disambiguating signature must be given.

7.7 Consequences of the theory—Lemmas

A theory is completely defined by that part of LSL that we have described so far, but it is usually convenient to have some of the consequences of the theory already identified as *lemmas* available for our use. Lemmas and axioms together make up the *theorems* of the theory. Penelope's syntax for traits includes an optional section for proving lemmas. Penelope does not force you to prove all lemmas, but does keep track of which lemmas are unproven.

The lemmas may be equations, references to traits, or schemes. If a trait is named as a lemma, the theory of this trait implies the theory of the named trait. Similarly, we can claim that the axioms of a theory are sufficient to prove some scheme, such as that lists are partitioned by *empty*, *head*, and *tail.*

⟨*consequences*⟩ ::=
  implies
    [[⟨*eq_part*⟩]]
    [[⟨*scheme*⟩]]*
    [[⟨*trait_ref*⟩]]*,

The ⟨*term*⟩ in a lemma must be of sort *Bool.* The only variables that may appear free in the term are those declared in the list of variables. No identifier may be used as the name of more than one axiom or lemma in any given trait.

7.8 Proof section

Proofs of lemmas are segregated into a proof section at the end of the trait. The theory that is available for proving a given lemma consists of the axioms, assumptions, and proved lemmas that precede the given lemma. LSL does not include a section of the trait for proving lemmas.

The proof section may contain proofs of theorems in other traits, although this would be unusual. Penelope prompts for a theorem name for each ⟨*named proof*⟩ and attempts to fill in the name of the trait automatically.

⟨*proof section*⟩ ::=
  --| proof section
  [[named_proof]]+
  --| end proof section
⟨*named_proof*⟩ ::=
  --| ⟨*identifier*⟩ [[**in trait** ⟨*identifier*⟩]] **:**
  [[⟨*optional_proof*⟩]]
⟨*optional_proof*⟩ ::=
  [[⟨*rewrite_annotations*⟩]]
  --| proof:
  ⟨*proof*⟩

You can create a proof section for a trait by clicking on `proof` in the help pane menu when the cursor is positioned at the trait. Penelope automatically calculates which consequences of the current theory remain to be proved. You can position the cursor at the *last* ⟨*named proof*⟩ of the proof section and click on `insert-obligations` to add any obligations not already present in the proof section.

Chapter 8

Simplification and proof: Penelope's proof editor

8.1 Introduction

Penelope includes a proof editor for simplifying preconditions and proving verification conditions and lemmas.

8.1.1 Sequents

Each statement to be proved or simplified is presented in the form of a *sequent*, a set of *hypotheses* and a *conclusion.* A sequent is often written in this manual in the form $\Gamma \Rightarrow P$, where Γ represents the hypotheses and P the conclusion. In Penelope a sequent is displayed with numbered hypotheses and the conclusion is indicated by >>. Penelope displays the sequent $n> = 0 \Rightarrow 0< = n$ as follows:

```
--!  1.(n>=0)
--!  >>(0<=n)
```

Note that all lines of a proof begin with the compound delimiter `--!`.

8.1.2 Available theory

Each proof in Penelope takes place in the context of an available theory. Within a compilation unit, the theory is determined by context clause annotations and all the local lemmas currently in force. The theory that is available for proving a given lemma consists of the axioms, assumptions, and proved lemmas that precede the given lemma.

8.1.3 Structure of proofs

Penelope's proofs are tree-structured. Each node of the tree corresponds to a sequent to be proved and one *proof step* that proves it, possibly subject to proving other, derivative sequents. For example, to prove the sequent $\Gamma \Rightarrow a$ **and** b, you can use a rule we call `and-synthesis`, which commits you to prove instead the derivative sequents $\Gamma \Rightarrow a$ and $\Gamma \Rightarrow b$. The children or *subproofs* of the node correspond to the further sequents needed to prove it. Leaves of a completed proof correspond to sequents that require no further proof (e.g., a sequent whose conclusion is "**true**"). While constructing a proof, the leaves also include unproved sequents.

Each proof step is an instance of one of Penelope's *proof rules*. When a proof step has two children, Penelope indents them for better readability.

If Penelope displayed every sequent in a proof tree, the buffer would be filled up with proofs. Penelope therefore displays only unproved sequents. If we do not wish to look at a sequent (perhaps because it is very long or we can't prove it yet), we can turn it into a diamond (`<>`) by selecting `hide-sequent` from the help-pane menu.[1]

8.1.4 Simplifying preconditions using the proof editor

We can use the proof editor for *simplification*. Penelope generates preconditions for executable code, including statements and declarations, as well as Penelope constructs representing the elaboration of library units (see Section 6.5.5). We can always examine the pre- or postcondition of such a construct by selecting `show-precondition` or `show-postcondition` from the help-pane when the cursor is positioned on the construct. We can also use the proof editor to simplify preconditions by selecting either `simplify-precondition` or `simplify-postcondition` from the help-pane. Such simplification makes preconditions easier to read and results in verification that is more likely to "replay"—that is, to still be valid even after changes to other parts of the program. What is left to prove after using the proof editor for simplification becomes the precondition of the simplification. The leaves of the simplification proof tree can be hidden by using the `hide-sequent` command discussed above.

```
⟨proof⟩ ::=
    --! <>
```

[1]It is possible to suppress all display of a given proof. When the cursor is positioned on a proof, issue the command `alternate-unparsing-toggle` from the Options menu. This suppresses the display of the selected proof. Be careful, though: if the buffer is written to a file in text form while the proof is suppressed, the proof will be lost.

8.1.5 The proof rules

Penelope's proof rules fall into several groups, which we discuss in most of the remaining sections of this chapter. For each proof rule, we give the help-pane menu name and its effect. For clarity, we sometimes give the effect of the rule mathematically, using the notation of the sequent calculus. The rule is written in a natural deduction style. Proving the sequent(s) above the line is sufficient to prove the sequent below the line. An example is the rule (thinning) that says we can remove an extra hypothesis:

$$\frac{n > 0 \Rightarrow abs(n) = n}{n > 0, n < 100 \Rightarrow abs(n) = n}$$

A terminal rule (one that requires no further proof) has no sequent above the line, e.g.:

$$\frac{}{\Gamma \Rightarrow \textbf{true}}$$

Unless otherwise noted, all of the sequents above the line must be proved. Appendix C contains a summary of the proof rules.

The large number of proof rules available may make the Penelope prover seem formidable. In fact, most people find it surprisingly easy to use once they become familiar with it. Penelope's proof steps fall into several basic groups: the application of automatic simplifiers or rewriters; the application of some available theorem (called instantiation); rules (such as `and-synthesis`, mentioned above) that break down the conclusion or hypotheses according to their syntactical form; and proof-structuring rules, such as proof by cases or proof by induction. The rules in this chapter are grouped according to these approaches.

Some of the proof rules are based on the syntax of the sequent. Such rules are fragile, in that minor changes to the program often change the syntax of preconditions or verification conditions, so that such proofs will not replay. We usually use more "logical" rules if possible. For example, proof by cases (`case` proof rule) is preferable to proof based on the fact that the conclusion of a sequent has the form **if-then-else** (`if-syn` proof rule). The logical structure of the sequent will survive small changes in the program, whereas the syntax of the sequent is less likely to.

Where the proof text is complex, or where the association between the help-pane menu item and the proof text is not obvious, we give the proof text for the rule, as well as the help-pane menu item.

8.1.6 Editing a proof

We cannot enter the text of proof rules into Penelope or edit them textually, as we would Ada code or a specification; we issue commands to implement proof steps. However, parts of the text are separate syntactic items (such as the number of a hypothesis); these items can be edited.

Unless otherwise noted, we can call on the proof rules when the cursor is positioned at a proof placeholder. The proof rules are organized with a hierarchical menu. When the cursor is positioned at a null proof, each item on the menu may represent a proof rule or a group of proof rules. For example, `thinning` is a proof rule, but `analyze-hypothesis` represents a group of proof rules. If the help-pane menu item corresponds to a single proof rule, clicking on the menu item causes the proof rule to be added to the proof tree. If a group of proof rules is chosen, a submenu appears with the individual rules in the group.

Two minor editing operations are possible at completed proof steps that have just one subproof:

- `delete-one-step` deletes the current proof step (the subproof remains).
- `swap-with-next-step` swaps the current proof step and its child. The subproof must have just one child.

8.2 Automatically applied rules

Penelope automatically applies the following trivial proof rules whenever possible. These rules do not appear on the help-pane. They all represent leaves of the proof tree, that is, completed proofs with no subproofs.

arithmetic

The conclusion is a theorem of arithmetic built in to Penelope.

conflicting-hypotheses

One hypothesis is the syntactic negation of another.

false-anal

false appears in the hypothesis list.

hypothesis

The conclusion appears in the hypothesis list.

self-identity

The conclusion has the form $x = x$.

true-syn

true is the conclusion.

8.3 Simplification

Penelope includes a number of simplification options that are grouped together. To use any of these, first select `simplify` from the proof help-pane menu. This places the cursor on ⟨*simplification_kind*⟩, whose help-pane menu shows a number of simplification rules.

More than one simplification step can be invoked at each simplification proof step. The simplification steps are applied in order. You can hit the `RETURN` key to enter a new simplification step. When the cursor is at a simplification step, you can click on `simplification-kind-list` to obtain a placeholder for a simplification step to be applied *before* the current one.

$$\langle \mathit{proof} \rangle ::= \\ \texttt{--!}\ \mathsf{BY}\ [[\langle \mathit{simplification\_kind} \rangle]]^{+}_{,}$$

$$\langle \mathit{proof} \rangle$$

SDVS-simplify, SDVS-simplify-conclusion

Penelope includes a Nelson-Oppen simplifier that is fairly good at quantifier-free predicate calculus, linear integer arithmetic, and real arithmetic (that is, arithmetic involving $+$, $<$, $=$, and multiplication by integer or real constants).

This simplifier is invoked by the commands `SDVS-simplify`

$$\langle \mathit{simplification\_kind} \rangle ::= \quad \mathsf{simplification}$$

or `SDVS-simplify-conclusion`

$$\langle \mathit{simplification\_kind} \rangle ::= \quad \mathsf{simplification\ of\ conclusion}$$

The effect is to replace the current sequent an equivalent (and usually simpler) sequent. The only difference is that `SDVS-simplify` will manipulate both the hypotheses and the conclusion, while `SDVS-simplify-conclusion` alters only the conclusion. Their results are logically equivalent,[2]but sometimes you want to keep the hypotheses rather than have the whole sequent recast.

[2]For logicians: over the base theory

limited-simplify

$\langle simplification\_kind \rangle$ ::= limited simplification

This proof rule has two distinct purposes. Its primary use is to induce rewriting (see Section 8.4). It is also a general simplifier for predicate calculus that is weaker than `SDVS-simplify`, but does not take as much time.

approximate-simplify, approximate-simplify-conclusion

This simplifier is analogous to the SDVS simplifier, and is invoked by `approximate-simplify`

$\langle simplification\_kind \rangle$ ::= approximate simplification

or `approximate-simplify-conclusion`:

$\langle simplification\_kind \rangle$ ::= approximate simplification of conclusion

These proof rules are useful for simplifying sequents involving approximate relational operators for the reals (see Section 4.4 and Table 4.2, page 26).

distribution

$\langle simplification\_kind \rangle$ ::= distribution

The SDVS simplifier is not good at distributing multiplication over addition. This proof rule performs that distribution.

array-simplification

$\langle simplification\_kind \rangle$ ::= array simplification

This proof rule expands all terms of the form *a[i=>v][j]* to the form **if** *i=j* **then** *v* **else** *a[j]*.

explicit-roundoff

$\langle simplification\_kind \rangle$ ::= explicit roundoff

Every instance of an "f-function" for an arithmetic function is replaced by an explicit expression involving rounding. For example, *fplus(x,y,z)* becomes

round_down(x)+round_down(y) <= *z* **and**
z <= *round_down(x)+round_down(y)*

prenex-simplify

$\langle$*simplification_kind*$\rangle$ ::= prenex simplification

In true prenex normal form, all quantifiers occur at the beginning of the formula. This simplification step attempts to put the conclusion of the sequent into a form closer to prenex normal form.

8.4 Rewriting

Penelope provides a limited automatic rewriting capability. That is, if we have a theorem $l = r$, we can ask Penelope to change instances of l to corresponding instances of r wherever they occur. In other words, we use the theorem as a *rewrite rule*. Penelope rewrites under the direction of the user and only does it once per user directive, so Penelope does not have the power (or the pitfalls) of automatic rewriting systems. In this section, we discuss what kinds of rewrite rules are available in Penelope, how to make a theorem a rewrite rule, and how to invoke rewriting. See also Section 8.5 for a discussion of using theorems to rewrite sequents just once.

8.4.1 Kinds of rewrite rules

Penelope can use theorems in the form of equations to rewrite one side of the equation to the other. Alternatively, instances of a theorem can be rewritten to **true**. Note that Penelope cannot use theorems of the form $p \rightarrow b = c$ as automatic rewrite rules, because in general we cannot assume that p holds.[3]

Form of the theorem	Rewrites	As
$l = r$	$l^{\vec{x}}_{\vec{v}}$	$r^{\vec{x}}_{\vec{v}}$
P , **forall** $\vec{v}$ :: P	$P^{\vec{x}}_{\vec{v}}$	**true**
not P , **forall** $\vec{v}$ :: **not** P	$P^{\vec{x}}_{\vec{v}}$	**false**

[3] P^x_y means P with x substituted for y. In $l^{\vec{x}}_{\vec{v}}$, substitution is simultaneous.

8.4.2 How to make a rewrite rule

There are three ways to make a rewrite rule. First, there are some theorems which we always want to apply as rewrite rules; for example *factorial(0) = 1* is a rewrite we always want to apply. The syntax for each axiom, lemma, and local lemma (see Section 6.3.2.3) includes an optional ⟨*rewrite_indication*⟩, **(rewrite)**. If present, a rewrite indication means that the axiom, lemma, or local lemma is always to be treated as a rewrite rule.

Usually, however, we want more control over rewriting. We use a *rewrite annotation* to indicate that within a certain scope a particular theorem is to be used as a rewrite rule. We can specify rewrite rules for a subprogram body, a package body, or the proof of a lemma in a trait.

⟨*rewrite_rule*⟩ ::=
  --! **rewrite rule:** ⟨*identifier*⟩ ⟨*trait_spec*⟩;

We can also specify a rewrite rule to apply during a proof (see page 83).

A rewrite rule invoked by a rewrite annotation in a subprogram body or package is *active* within the entire subprogram body or package. A rewrite rule specified before a proof is active within the entire proof. A rewrite rule specified in a proof step is active in all subproofs of that proof step.

8.4.3 How to invoke rewriting

We invoke rewriting by using the `limited-simplify` proof rule (see page 77).

⟨*simplification_kind*⟩ ::= limited simplification

Each `limited-simplify` proof step attempts to apply all the active rewrite rules, in some unspecified order.

Penelope applies rewrite rules only on command, and then applies each rule just once. Thus Penelope's rewriting always terminates. Multiple rewrites may produce more simplification than a single rewrite. Note that `limited-simplify` may rewrite hypotheses.

8.5 Instantiation of mathematical theorems

The mathematical part of the specification of the program contains axioms and lemmas that are useful for verification. The proof rules in this section are used to apply a theorem to

a sequent or to make them active as rewrite rules. To use any of these rules, first select `instantiation` from the proof help-pane menu.

$\langle$*simplification_kind*$\rangle$ ::=
--! BY instantiation of $\langle$*identifier*$\rangle$ [[$\langle$*trait_spec*$\rangle$]] [[$\langle$*substitution_clause*$\rangle$]]
--! [[sideproof]]
--! $\langle$*instantiation_action*$\rangle$
$\langle$*proof*$\rangle$

A theorem is uniquely identified by the name of a trait and the name of the theorem within the trait (see Chapter 7) or by the name of a local lemma (see Section 6.3.2.3). Given the name of a theorem, Penelope automatically fills in the name of the theory, if possible. We can override a name that Penelope picks by editing the non-terminal $\langle$*trait_spec*$\rangle$ in the text for a proof rule.

$\langle$*trait_spec*$\rangle$ ::=
in trait $\langle$*identifier*$\rangle$
| , a local lemma,

A theorem is typically a universal mathematical statement—i.e., of the form, "for all integers x and y, ..." To apply such a theorem we often have to instantiate it for particular values of x and y. All appeals to theorems of the available theory are called instantiations. The following syntax is used in instantiating theorems:

$\langle$*substitution_clause*$\rangle$ ::=
with ($[[\langle \textit{pointwise\_substitution} \rangle]]^{+}_{;}$)
$\langle$*pointwise_substitution*$\rangle$ ::=
$\langle$*term*$\rangle$ for $\langle$*sorted variable*$\rangle$

This $\langle$*substitution_clause*$\rangle$ says to (simultaneously) substitute each term for the corresponding variable of the theorem. Penelope tries to supply the substitution if you do not and if the instantiation is being used for rewriting. If Penelope is not sure how to instantiate the theorem, it prompts you with the names of the theorem's free variables so that you can enter the pointwise substitution. You may override Penelope's default substitution by filling in the substitution clause yourself. (Of course, that means that if you change your mind and want Penelope to supply the substitution, you have to delete your substitution clause.)

The different proof rules in this section apply a theorem in different ways, indicated by an *instantiation action*. We can either add the theorem to the hypothesis list or use it to rewrite the conclusion of the current sequent. We name each of the proof rules by the help-pane

menu item for its instantiation action. Help-pane menu items enable us to switch from one of these proof rules to any of the others.

Sideproofs are used to prove conditional rewriting rules. See the discussion below, under `rewrite-left-to-right`.

rewrite-left-to-right

⟨*instantiation_action*⟩ ::=
rewriting left to right

If an axiom or lemma is in the form of an equation or conditional equation, we can substitute for its free variables and substitute the left side of the equation for the right everywhere in the sequent. Equations may be of any of the following forms:

$$\begin{aligned} l &= r \\ c \rightarrow l &= r \\ l &= (\textbf{if } c \textbf{ then } l \textbf{ else } r) \\ l &= (\textbf{if } c \textbf{ then } r \textbf{ else } l) \end{aligned}$$

In each case (l) is substituted for (r). For the conditional equations (the last three cases), a *sideproof* is created to discharge the condition c.

For example, if we have a theorem *pop(push(e,s))* $= s$, we can substitute $a[5]$ and *abs_stack(r)* for e and s, and then replace, for example,

$$stack\_sum(pop(push(a[5], abs\_stack(r))))$$

by $stack\_sum(abs\_stack(r))$.

More formally, if $c \rightarrow l = r$ is a theorem with free variables $\vec{v}$, we can substitute $r' = r^{\vec{x}}_{\vec{v}}$ for $l' = l^{\vec{x}}_{\vec{v}}$, if $c^{\vec{x}}_{\vec{v}}$ holds.[4] We have

$$\frac{\Gamma \Rightarrow c^{\vec{x}}_{\vec{v}}; \quad \Gamma, \Gamma^{r'}_{l'} \Rightarrow Q^{r'}_{l'}}{\Gamma \Rightarrow Q}$$

The mathematics for the other forms of `rewrite-left-to-right` is defined analogously.

[4] P^x_y means P with x substituted for y. In $l^{\vec{x}}_{\vec{v}}$, substitution is simultaneous.

When the theorem is in the form of a conditional equation ($c \rightarrow l = r$), a help-pane item `establish-condition` appears on the help-pane menu. Clicking on this item positions us at the proof of $c^{\vec{x}}_{\vec{v}}$.

⟨*sideproof*⟩ ::=
--! [[establishing
  ⟨*proof*⟩
--! THEN]]

Section 8.4 describes a facility for defining rewrites of the first kind ($l = r$ unconditionally) that we always want to apply.

rewrite-right-to-left

⟨*instantiation_action*⟩ ::=
  rewriting right to left

This rule is just like `rewrite-left-to-right`, except that the left side of the equation is substituted for the right.

rewrite-to-true

⟨*instantiation_action*⟩ ::=
  rewriting to true

If P is a theorem with free variables $\vec{v}$, we can substitute **true** for all instances of $P^{\vec{x}}_{\vec{v}}$, provided that the sorts of $\vec{x}$ match the sorts of $\vec{v}$.

Thus, we can replace

pop(push(a[5],abs_stack(r)))=abs_stack(r)

in the conclusion by **true**. Mathematically, we have:

$$\frac{\Gamma \Rightarrow Q^{\mathbf{true}}_{P^{\vec{x}}_{\vec{v}}}}{\Gamma \Rightarrow Q}$$

If the theorem has the form $c \rightarrow P$, a sideproof (see page 81) discharges c.

add-as-hypothesis

⟨*instantiation_action*⟩ ::=
as new hypothesis

When the syntax of a theorem does not make it well suited for either `rewrite-left-to-right` or `rewrite-to-true`, we can still instantiate it and enter it into the list of hypotheses. Thus, if P is a theorem with free variables $\vec{v}$, we have

$$\frac{\Gamma, P^{\vec{x}}_{\vec{v}} \Rightarrow Q}{\Gamma \Rightarrow Q}$$

add-as-rewrite-rule

⟨*instantiation_action*⟩ ::=
as rewrite rule

We can add an instantiated theorem to the list of rewrite rules. This modifies the effect of rewriting (see Section 8.4). Note that rewrite rules must be of the form $l = r$; the (instantiated) right side is substituted unconditionally for the left.

add-as-reversed-rewrite-rule

⟨*instantiation_action*⟩ ::=
as rewrite rule

This rule is like the previous one, except that an instantiated theorem of the form $l = r$ produces a rewrite rule $r = l$. That is, the left side of the theorem is substituted for the right.

forward-chain

⟨*instantiation_action*⟩ ::=
forward chaining [[⟨*integer*⟩]]

If the instantiated theorem is of the form

$$\bigwedge_i P_i\vec{x} \rightarrow Q\vec{x}$$

then forward chaining may be used. If there is some $\vec{v}$ such that *all* of $P_i\vec{v}$ are in the hypothesis list, then $Q\vec{v}$ is added to the hypothesis list.

By default, a single pass is made over the hypothesis list. By specifying an ⟨*integer*⟩ n you can cause Penelope to make n passes over the hypothesis list. For example, if $f(m) \rightarrow f(m+1)$ is the theorem being instantiated, and $f(0)$ is in the hypothesis list, then two passes add $f(1)$ and $f(2)$ to the hypothesis list.

The ⟨*integer*⟩ is called the *bound* on forward chaining. You can increase the bound by using the `increase-bound` command on the help-pane menu.

disable-rewrite-rule

> ⟨*instantiation_action*⟩ ::=
> disabling rewrite rule

We can disable an instantiated rewrite rule for a particular subproof. Reversed rewrite rules cannot be disabled in this version of Penelope.

8.6 Proof-structuring rules

The proof rules discussed in this section are used to structure the proof: proof by cases, by contradiction, etc. Except for the thinning rule these rules tend to be relatively *robust*, in the sense that minor changes to the program are not apt to change their applicability.

See also Section 8.9 for proofs by induction and Section 8.10 for proofs by extensionality.

case

> ⟨*proof*⟩ ::=
> --! BY cases, using ⟨*term*⟩
> --! CASE TRUE [[, then rewriting]]
> ⟨*proof*⟩
> --! CASE FALSE
> ⟨*proof*⟩

The ⟨*term*⟩ must be boolean. Two cases are considered: the term is true or it is false. Because this rule is robust, it is preferable to `if-syn` (see Section 8.7). If the phrase then rewriting is present, instances of the ⟨*term*⟩ are replaced in the first subproof by **true**, in the second by **false**.

claim

> ⟨*proof*⟩ ::=
> --! BY claiming ⟨*term*⟩

$\langle proof \rangle$
--! THEN
$\langle proof \rangle$

The term must be boolean. The first subproof establishes the claim; the second uses it to prove the original conclusion.

contradiction

For a proof by contradiction, first select `contradiction` from the help-pane menu.

$\langle proof \rangle$ ::=
--! BY contradiction[[$\langle optional\_hypothesis \rangle$]]
$\langle proof \rangle$
$\langle optional\_hypothesis \rangle$ ::= , in $\langle integer \rangle$

This proof rule has two forms. In the first form, we assume the conclusion does not hold, and prove **false**. To invoke this form of contradiction, delete the $\langle optional\_hypothesis \rangle$ placeholder.

A second form of proof by contradiction is to assume that the conclusion does not hold and try to disprove one of the hypotheses. For this form of proof by contradiction, fill in the number of the hypothesis to be disproved in the $\langle optional\_hypothesis \rangle$ placeholder.

thinning

$\langle proof \rangle$ ::=
--! BY thinning $\langle hypotheses\ to\ be\ thinned \rangle$

$\langle proof \rangle$ $\langle proof \rangle$ ::=
$[[\langle integer \rangle]]^+_,$
| all
| all but $[[\langle integer \rangle]]^+_,$

We can remove unneeded hypotheses, usually to improve readability. Also, some hypotheses result in complex sequents after "simplification" by the SDVS simplifier. If such a hypothesis (usually an implication or if-then-else) is unneeded, it can be removed. The integers refer to hypothesis numbers, making this a fragile step.

8.7 Rules based on the syntax of the conclusion

Rules whose names end in -synthesis (or -syn) are based on the syntax of the conclusion, specifically on its major connective. There is one for each connective, and each does the obvious thing. The name of the rule (e.g. `and-syn`) and the corresponding text (e.g., BY synthesis of AND) recall the connective. Note that syntactically-based rules are fragile, in that minor changes to the program often change the syntax of preconditions or verification conditions, so that such proofs will not replay. Nevertheless, there are times when we have to dig into the syntax of the sequent in order to simplify or prove it, especially when a quantified term is embedded in a hypothesis or conclusion.

To invoke a rule based on the syntax of the conclusion of the current sequent, click on `synthesize-conclusion` on the help-pane menu. The applicable synthesis rules will appear on a submenu. You can also execute the special command `!s` (not on the help-pane menu) to select a rule; it does not make its selection intelligently, but just picks the simplest rule for the major connective of the conclusion.

See also Section 8.9 on proof by induction (for sequents with a universally quantified conclusion) and Section 8.10 on proof by extensionality (for sequents with a conclusion in the form of an equality).

exists-syn

> ⟨*proof*⟩ ::=
> --! BY synthesis of EXISTS [[exhibiting ⟨*term*⟩]]
> ⟨*proof*⟩

We prove that a value exists by producing a term for it. The resulting sequent replaces the bound variable of the quantifier by our suggested term (called a *witness*). If the witness is omitted, Penelope attempts to supply it by matching with available hypotheses.

forall-syn

The bound variable is replaced by a fresh free variable.

forall/implies-syn

Sometimes Penelope generates preconditions that include nested instances of **forall** and `->`. This rule successively replaces universally quantified variables by fresh free variables and places the antecedents in the hypothesis list. For example, given a sequent with no hypotheses and the conclusion

$$\forall i : Int :: (p \rightarrow (\forall j : Int :: q \rightarrow i = j))$$

this rule produces the sequent $p, q \Rightarrow i = j$.

Note that the `forall/implies` proof rule is somewhat more robust than use of the `forall` or `implies` proof rules.

affirmation-synthesis

We can replace a conclusion of the form $Q = \textbf{true}$ with Q.

and-syn

We can prove a conclusion $Q_1 \wedge \ldots \wedge Q_N$ by proving each Q_i separately.

It sometimes happens that a change in the program causes the number of subproofs to become unequal to the number of conjuncts in the conclusion. In this case an error message appears: "wrong number of subproofs". Click on the proof (or on the error message). The help-pane menu displays **and-syn**. Click on **and-syn** and Penelope will attempt to adjust the subproofs of the current proof step: an empty proof at the end will be deleted if there are too many proofs; an empty proof will be created at the end if there are not enough proofs. You may have to do some editing to match proofs with sequents.

denial-synthesis

We can replace a conclusion of the form $Q = \textbf{false}$ with $\neg Q$.

equals-synthesis

This rule applies to a conclusion of the form $P = Q$ where P and Q are both boolean. It reduces the proof of the equation to proving each using the other as an additional hypothesis.

if-branch-selection

```
⟨proof⟩ ::=
   --! BY establishing IF condition ⟨term⟩
  ⟨proof⟩
--! THEN
  ⟨proof⟩
```

This rule creates two subproofs. The first attempts to prove the term. The second conclusion is formed by substituting **true** for every occurrence of the term in the original sequent.

The `SDVS-simplify` proof rule (see page 76) is more robust.

if-pair

Suppose the conclusion of a sequent is of the form

$$\textbf{if } P \textbf{ then } Q \textbf{ else } R$$

and one hypothesis is of the form

$$\textbf{if } P \textbf{ then } A \textbf{ else } B$$

We try to prove $A \wedge P \to Q$ and $B \wedge \neg P \to R$.

The `SDVS-simplify` proof rule is more robust.

if-syn

The conclusion is of the form **if** P **then** R **else** S We make two subproofs, one for P and one for $\neg P$. The `case` proof rule is more robust.

not-equals-synthesis

We replace a conclusion of the form $x \neq y$ by $\neg(x = y)$.

not-syn

We prove a conclusion $\neg Q$ by contradiction. That is, we assume Q and prove **false**. See page 85.

or-syn-l

If the conclusion is of the form $Q_1 \vee Q_2$, we can prove Q_1, assuming Q_2 does not hold.

or-syn-r

If the conclusion is of the form $Q_1 \vee Q_2$, we can prove Q_2, assuming Q_1 does not hold.

xor-syn

If the conclusion is of the form $Q_1 \oplus Q_2$, we can replace it with $(\neg Q_1 \wedge Q_2) \vee (Q_1 \wedge \neg Q_2)$.

8.8 Rules based on the syntax of a hypothesis

Rules whose names end in -analysis (or -anal) are based on the syntax of one of the hypotheses. The corresponding text of the proof is similar:

BY analysis of IMPLIES [[in ⟨*integer*⟩]]

where the integer gives the number of the intended hypothesis.

As noted in the introduction to this chapter, it is usually preferable to avoid syntax-based proof rules, because the syntax of a precondition or verification condition can be radically changed by a slight modification to the program. Nevertheless, there are times when we must dig into the syntax of the sequent in order to simplify or prove it, especially when a quantified term is embedded in a hypothesis or conclusion. The only proof rules available in Penelope for quantified terms are syntactic ones.

Some analysis rules are followed by an optional **then thinning** clause. If present, this indicates that a rewritten hypothesis is to be removed because it is no longer needed. Otherwise Penelope avoids weakening the hypotheses of a sequent.

Sometimes `-analysis` rules that require a hypothesis number fail to "replay" because the hypothesis number has changed (a new hypothesis has been introduced); they can then be fixed by updating the hypothesis number in the proof.

If the hypothesis number is omitted, Penelope chooses the first hypothesis to which the rule is applicable.

To invoke a rule based on the syntax of a hypothesis of the current sequent, click on `analyze-hypothesis` on the help-pane menu.

axiom-of-choice

⟨*proof*⟩ ::=
  --! BY axiom of choice [[in ⟨*integer*⟩]]
  ⟨*proof*⟩

When a hypothesis has the form $\forall x : S \exists y : T :: P(x, y)$, then we may add a skolemized version: $\exists f : S \to T :: \forall x : S :: P(x, f[x])$ That is, f has sort $\mathbf{map}[S]\mathbf{of}T$.

exists-anal

⟨*proof*⟩ ::=
  --! BY analysis of EXISTS [[in ⟨*integer*⟩]] [[, then thinning]]
  ⟨*proof*⟩

By hypothesis, a certain value exists. Penelope produces a fresh constant to represent that value. If a hypothesis number is given, Penelope chooses the first existentially quantified variable in that hypothesis.

If no number is given, Penelope selects *all* existentially quantified hypotheses and produces fresh variables for all of the variables bound by those quantifiers. Note that this form of the proof rule avoids referring to a hypothesis number in the proof, and may hence be somewhat more robust than if a hypothesis number is given, since hypothesis numbers may change if the program or proof is modified.

If the then thinning clause is present, Penelope replaces the original hypothesis(es) with the new one(s).

forall-anal

$\langle proof \rangle$ ::=
  --! BY analysis of FORALL [[in $\langle integer \rangle$]]
  --! WITH $[[\langle term \rangle]]^+_,$ FOR $[[\langle sorted\ variable \rangle]]^+_,$
--! [[, then thinning]]
  $\langle proof \rangle$

We can instantiate a universal hypothesis. We have to provide the desired instance. If the then thinning clause is present, Penelope replaces the original hypothesis(es) with the new one(s).

equals-analysis

$\langle proof \rangle$ ::=
  --! BY $\langle side\_of\_equation \rangle$ substitution
  --! [[of $\langle integer \rangle$]] [[, then thinning]]
  $\langle proof \rangle$
$\langle side\_of\_equation \rangle$ ::= left | right

Sometimes we want to rewrite the conclusion of a sequent using a hypothesis as a rewrite rule. If $l = r$ is a hypothesis, we can substitute r for all free occurrences of l everywhere in the conclusion and append hypotheses that result from substituting for l in all the hypotheses. By default the left side of the equation is replaced by the right, but by changing the *side_of_equation* we can request it the other way around. Note that this rewriting has nothing to do with the rewriting discussed in Section 8.4. This rule causes the specified hypothesis, and not the active rewrite rules, to be used just once for rewriting the conclusion. This rule does not change the set of active rewrite rules.

For left substitution we have:

$$\frac{\Gamma, l = r, \Gamma_l^r \quad \Rightarrow Q_l^r}{\Gamma, l = r \quad \Rightarrow Q}$$

If the **then thinning** clause is present, Penelope replaces the original hypothesis(es) with the new one(s).

and-anal

We can replace a hypothesis $P_1 \wedge \ldots \wedge P_n$ by n hypotheses, $P_1, \ldots, P_n$.

if-else-anal

One of the hypotheses is of the form **if** P **then** R **else** S. We claim that $\neg P$ holds (first subproof) and then use P and S as hypotheses.

if-then-anal

One of the hypotheses is of the form **if** P **then** R **else** S. We claim that P holds (first subproof) and then use P and R as hypotheses.

imp-anal

If $P_1 \rightarrow P_2$ is a hypothesis, we can prove P_1 and then use P_2 as a hypothesis.

not-equals-analysis

We replace a hypothesis of the form $x \neq y$ by $\neg(x = y)$.

not-analysis

If a hypothesis is of the form $\neg P$, we attempt to prove P.

or-anal

If $P_1 \vee P_2$ is a hypothesis, we can prove two sequents, one using P_1 and one using P_2.

xor-anal

If a hypothesis is of the form $P_1 \oplus P_2$, we can replace it with $(\neg P_1 \wedge P_2) \vee (P_1 \wedge \neg P_2)$.

8.9 Proof by induction

Induction can be applied when the conclusion begins with some sequence of universal quantifiers. The proof step is

```
⟨proof⟩ ::=
   --! BY induction on ⟨variable⟩ [[with scheme ⟨integer⟩]]
  ⟨proof⟩
```

This proof step is not applicable unless the conclusion to be proved is a universally quantified formula, e.g.

$$\textbf{forall } x : S \ :: \ \textbf{forall } y : T \ :: \ P$$

We have to specify a bound variable to be the induction variable—in this case x or y. (By default, it is assumed that induction is over the first bound variable.) In addition, the theory must contain one or more induction schemes for the sort of the bound variable chosen (in this example, S or T). If the variable's sort has more than one possible induction scheme, we can specify which induction scheme to use. The induction schemes are structural induction (made available by a **generated by** clause for the sort) and complete induction (made available by a well-founded relation on the sort).

For example, stacks are generated by *empty* and *push*. To prove

$$\textbf{forall } s : \mathit{Stack} \ :: \ P(s)$$

by structural induction we are asked to prove both *P(empty)* and $P(s) \rightarrow P(\mathit{push}(e, s))$.

In complete induction over the well-founded relation "less than", the induction hypothesis is that some property holds for all elements of a sort "less than" some element x, and we must show that it therefore holds for x as well. Induction over *Int* is a special case of induction over a well-founded relation—namely, the relation "$\mathbf{abs}(x) < \mathbf{abs}(y)$". To prove **forall** $x : \mathit{Int}\ P(x)$ by induction we are asked to prove

$$\textbf{forall } x : \mathit{Int} :: (\textbf{forall } y : \mathit{Int} :: \mathbf{abs}(y : \mathit{Int}) < \mathbf{abs}(x : \mathit{Int}) \rightarrow P(y)) \rightarrow P(x)$$

Proofs by induction and extensionality (see Section 8.10) are rarely appropriate for verification conditions. Statements complex enough to require these proof techniques should be proposed as mathematical lemmas and referred to during proofs of verification conditions (see Section 8.5).

8.10 Proof by extensionality

Proof by extensionality is applicable when the conclusion of a sequent is of the form $t_1 = t_2$, and when we have available a **partitioned by** scheme for the sort S of the terms t_i.

The scheme provides a list of operators $op_1, \ldots, op_n$ that partition S. The extensionality rule obtained from this scheme says that to show that t_1 and t_2 are equal, it suffices to show that t_1 and t_2 cannot be distinguishable by using $op_1, \ldots, op_n$. For example, stacks are partitioned by the functions *is_empty*, *pop* and *top* (see Section 7.6.4). So to show that two stacks s_1 and s_2 are equal, we can show that these observer functions cannot distinguish between them:

$$is\_empty(s_1) = is\_empty(s_2) \wedge top(s_1) = top(s_2) \wedge pop(s_1) = pop(s_2)$$

If there is more than one **partitioned by** scheme for the sort of the t_i, then we can enter a number for the scheme (**partitioned by** schemes are numbered starting with 1). The number is optional if there is just one scheme.

Formally, we say that x and y are indistinguishable using op if when x and y are each substituted for w in a term of the form $op(\ldots, w, \ldots)$ the values are equal. Formally, let

$$Ind(op, x, y) = \forall a_1, ..., a_n(\bigwedge_{sort(a_i)=S} t^x_{a_i} = t^y_{a_i})$$

where $t = op(a_1, \ldots, a_n)$. Then the extensionality rule is

$$\frac{\Gamma \Rightarrow \bigwedge_i Ind(op_i, x, y), \text{where x,y of sort S, partitioned by } op_1, \ldots, op_n}{\Gamma \Rightarrow (x = y)}$$

8.11 Seldom-used rules

The following rules are retained for compatibility with older versions of Penelope.

direct-subst

$\langle proof \rangle$ ::=
  --! BY substitution of TRUE [[for $\langle integer \rangle$]]
  $\langle proof \rangle$

True is substituted in the conclusion for all occurrences of the numbered hypothesis. (It is usually simpler to use the SDVS simplifier—see page 76.)

assume

We can select **assume** from the help-pane menu to bypass Penelope's prover. This rule is almost never needed. In a future version of Penelope this rule will be eliminated.

8.12 Interface to the HOL theorem prover

Penelope includes a simple interface to the HOL theorem prover. This is an experimental feature to demonstrate the feasibility of interfacing Penelope to other tools for building theories and proving theorems.

To translate a theory to HOL format, we first need to have it in a Penelope buffer. We then invoke the `write-hol` command from the Penelope menu (click on the right mouse button to get the Penelope menu). This command creates a dialog box that requests the name of the trait to translate and a directory to receive the resulting file. The file name is constructed by appending `.ml` to the name of the trait (e.g., `Stack.ml`). Each trait must be separately translated. The format of the output files is suitable for use with HOL-88.

For further information about the HOL theorem prover, see [4].

Appendix A

Verification of a stack package

This appendix contains a complete verification of a generic stack package that defines a type of stacks. The verification begins with traits providing the mathematics for our implementation:

- *Lists*—see Figure 7.1 on page 61 for the trait for *Lists*
- *Stacks*
- *StackImpl*—implementation of stacks by records

The proofs of the lemmas in the traits are omitted for brevity.

Following the mathematics, we present the generic stack package, which introduces a private type `stack`, with operations `empty`, `push`, and `pop`, as well as the exceptions `stack_full` and `stack_empty`.

A.1 Trait *Stacks*

A stack is mathematically the same as a list. The essence of both is a LIFO discipline. The functions have different names.

A.2 Trait *StackImpl*

In order to implement stacks, we have to represent them in terms of Ada data structures. We choose to represent a stack by a record with two fields. Field *r.top* represents the current depth and field *r.contents* is an array indexed by integers; *r.contents[n]* represents the nth element placed on the stack. The axiom defining the *abs_stack* abstraction function expresses the representation. The representation invariant *inv_stack* is that *r.top* must lie between 0

Stacks: **trait**

includes $(Lists)(Stack$ for $List, Element$ for E,
$empty$ for $nil, push$ for $cons, top$ for $head$,
pop for $tail, size$ for $length)$
introduces
lemmas : $\forall s : Stack, i : Element$
top : $(top(push(i, s)) = i)$
pop : $(pop(push(i, s)) = s)$
empty : $(push(i, s) \neq empty())$
size0 : $(size(empty()) = 0)$
size : $(size(push(i, s)) = (1 + size(s)))$

Figure A.1: The trait *Stacks*

and *stack_limit*. We do not define the value of *stack_limit* here, because we want *StackImpl* to be reusable for implementations of stacks with different maximum depths. We do state that it must be greater than zero.

We also introduce lemmas (*top*, *pop*, *size*, *empty*, and *push*) that show how the mathematical operations on stacks are translated to operations on records. That is, if r represents a stack, we show how record operations represent operations on *abs_stack(r)*. The proofs of the lemmas show that the translations are correct, given our chosen representation of stacks. Additional lemmas (*dont_care*, and *dont_care_helper*) are needed to prove the other lemmas.

Note that requiring *inv_stack* to hold for lemma *push* is actually too strong. Requiring *r.top >= 0* would have been sufficient. Note also that we don't need to require that $r.top < stack\_limit$.

A.3 Stack package—The declaration

Our generic stack package has two generic parameters: `max`, the maximum depth of the stack, and `elem`, the type of objects on the stack. The package introduces the type stack (`st`), operations on the stack, and two exceptions, `stack_empty` and `stack_full`. We declare type `stack` to be private, based on sort *Stack*. We will use abstraction in discussing the effect of the various operations on type `stack`, rather than annotating and verifying them in terms of the implementation. Abstraction makes the specifications more readable and also makes them reusable if the implementation changes. Further, verification of clients of package `stack` will be in terms of abstract stacks, rather than in terms of a particular implementation; this not only simplifies the verification, but also insulates it against changes in the implementation of the package. The maximum stack depth is `stack_limit`, which is the same as `max`. We use the function, but we could have used `max` as a global instead.

A generic stack package would ordinarily be written so that each instantiation created a

$StackImpl$: **trait**

sort $StackRec$ **is record;**
$top : Int$,
contents: **array**$[Int]$ **of**$Element$
end record
includes $(Stacks)$
introduces
$abs\_stack : StackRec \rightarrow Stack$
$inv\_stack : StackRec \rightarrow Bool$
$stack\_limit : \rightarrow Int$
asserts
$\forall s : StackRec, i : Int, st : Stack, e : Element$
abs_stack : $(abs\_stack(s) = (\text{if } (s.top = 0) \text{ then } empty()$
$\quad \text{else } push((s.contents[s.top]), abs\_stack(s[.top \Rightarrow (s.top - 1)]))))$
inv_stack : $(inv\_stack(s) = ((0 \leq s.top) \wedge$
$\quad (size(abs\_stack(s)) \leq stack\_limit)))$
non_trivial : $(stack\_limit() > 0)$
implies
$\forall n : Int, s : StackRec, i, x : Int, y : Element$
dont_care_helper : $((((x > s.top) \wedge (n = s.top)) \wedge (s.top \geq 0)) \rightarrow$
$\quad (abs\_stack(s[.contents \Rightarrow (s.contents[x \Rightarrow y])]$
$\quad\quad [.top \Rightarrow s.top]) = abs\_stack(s)))$
dont_care : $(((x > s.top) \wedge (s.top \geq 0)) \rightarrow$
$\quad (abs\_stack(s[.contents \Rightarrow (s.contents[x \Rightarrow y])]$
$\quad\quad [.top \Rightarrow s.top]) = abs\_stack(s)))$
top : $((inv\_stack(s) \wedge (s.top > 0)) \rightarrow$
$\quad (top(abs\_stack(s)) = (s.contents[s.top])))$
pop : $((inv\_stack(s) \wedge (s.top > 0)) \rightarrow$
$\quad (pop(abs\_stack(s)) = abs\_stack(s[.top \Rightarrow (s.top - 1)])))$
empty : $((abs\_stack(s) = empty()) = (s.top = 0))$
size : $((s.top \geq 0) \rightarrow (size(abs\_stack(s)) = s.top))$
push : $(inv\_stack(s) \rightarrow (push(y, abs\_stack(s)) =$
$\quad abs\_stack(s[.top \Rightarrow (s.top + 1)][.contents \Rightarrow$
$\quad\quad (s.contents[(s.top + 1) \Rightarrow y])])))$

Figure A.2: Mathematics for stack implementation

stack, rather than a type, as is the case here. We cannot, however, use variables inside the package body to represent the stack, since Penelope does not yet provide the necessary support.

```
--| with trait StackImpl;
generic
  type elem is private;
    --| based on Element;
  max: in integer;
  --| lemma pos_stack: (max=stack_limit());
package stack is
  stack_full : exception;
  stack_empty : exception;
  type stack is private ;
    --| based on Stack;
```

A.3.1 The function stack_limit

The function stack_limit returns the maximum depth of the stack. Note that in the annotation *stack_limit()* refers to the constant from trait *StackImpl100*, not to the function being specified. We use this function to simulate an Ada constant. Alternatively, we could have used a variable.

```
function stack_limit return integer;
  --| where
  --|    global max: in;
  --|    return stack_limit();
  --| end where;
```

A.3.2 The function empty

The function empty returns a new, empty stack. It does not raise any exceptions. Empty includes a **return** annotation because it is a function. The abstraction of the value returned is the stack *empty*(). The empty parentheses indicate that this value is a constant.

```
function empty return stack;
  --| where
  --|    return s such that s=empty();
  --| end where;
```

A.3.3 The function `is_empty`

The function `is_empty` tells us whether the stack is empty. It does not raise any exceptions. The value returned corresponds to the function *is_empty* in trait *Stacks.*

```
function is_empty return boolean;
  --| where
  --|   return (empty(s));
  --| end where;
```

A.3.4 The procedure `push`

The procedure `push` modifies `s` by pushing `n` onto it. We require *inv_stack(s)* on entry to the procedure and maintain it on normal termination.

Since `push` is a procedure, we use **out** annotations rather than a **return** annotation. Note that the out annotation for `push` has to distinguish between `s` on exit from the procedure and its value on entry (**in** *s*). We do not have to say "**in** *n*" because the value of *n* is not changed by the procedure.

There is no confusion between the Ada procedure `push` and the mathematical function *push*—in annotations the mathematical function is always intended.

```
procedure push(n : in elem; s : in out stack);
  --| where
  --|   out (s=push(n,in s));
  --|   raise stack_full <=>
      in (size(s)=stack_limit);
  --| end where;
```

The out annotation describes the result of normal termination. The propagation constraint states that, if the procedure terminates, then it will terminate by raising the exception `stack_full` if and only if the depth of the stack is *stack_limit* on entry to the procedure. Given the *inv_stack*, this assertion is equivalent to the apparently weaker *size(s)>=stack_limit.*

A.3.5 The procedure `pop`

```
procedure pop(n : out elem; s : in out stack);
  --| where
  --|   out (n=top(in s));
  --|   out (s=pop(in s));
  --|   raise stack_empty <=> in (s=empty());
```

```
--| end where;
```

The procedure pop modifies s by removing an element from it and returning that element as n. Note that both mathematical functions *pop* and *top* are needed to describe the effect of Ada procedure pop. The exact propagation annotation states that (assuming the procedure terminates) it will terminate by propagating stack_empty if and only if it is called with an empty stack.

A.3.6 The private part

In the private part of the package, we identify the abstraction function and representation invariant for type stack.

```
private
  type cont is array(integer) of elem;
  type stack is record
    top:  integer;
    contents:  cont;
  end record;
    --| abstraction function : abs_stack;
    --| representation invariant: inv_stack;
end stack;
```

A.4 Stack package—The body

A.4.1 Proof of function stack_limit

Penelope repeats the subprogram annotation from the subprogram declaration. The function is verified based on the lemma asserted with the declaration of max. Any instantiation of the stack package must show that the value for max satisfies restrictions placed on *stack_limit()*.

```
package body stack is
  function stack_limit return integer
    --| where * * *
    --|    return stack_limit();
    --| end where;
    --!   rewrite rule: pos_stack in local lemmas;
    --! VC Status: proved
    --! BY synthesis of TRUE
  is
  begin
    return max;
  end stack_limit;
```

A.4.2 Proof of function empty

The proof of the empty function uses the definitions of *inv_stack* and *abs_stack* as rewrite rules to show that setting *s.top* to 0 results in a record that satisfies the representation invariant and represents the empty stack. Theorem *non_trivial* is only needed to show that *stack_limit* is greater than zero.

Penelope automatically translates the abstract specification into a concrete form for the implementation. Each instance of a variable of type stack now refers to the implementation type. The abstraction function (*abs_stack*) is applied to variables of type stack. Input arguments of type stack are required to satisfy the representation invariant (*inv_stack*) and output arguments are guaranteed to satisfy it.

This translation does not appear in the annotation of the body, because Penelope merely copies the annotation of the subprogram declaration, but it does appear in statement preconditions and verification conditions. The use of rewrite rules here causes Penelope to expand those functions automatically.

```
function empty return stack
  --| where * * *
  --|   return z such that (z=empty());
  --| end where;
  --!   rewrite rule: abs_stack in trait StackImpl;
  --!   rewrite rule: inv_stack in trait StackImpl;
  --! VC Status: proved
  --! BY instantiation of non_trivial in trait StackImpl as new hypothesis
  --! BY simplification
  --! BY synthesis of TRUE
is
  temp :  stack;
begin
    temp.top:=0;
    return temp;
end empty;
```

A.4.3 Proof of function is_empty

The function empty can be verified entirely by simplifying the precondition of its only statement. We use rewriting to reduce the precondition of the return statement to true.

```
function is_empty(s :  in stack) return boolean
  --| where * * *
  --|   return (s=empty());
  --| end where;
```

```
  --!    rewrite rule: abs_stack in trait StackImpl;
  --! VC Status: proved
  --! BY synthesis of TRUE
is
begin
  --! BY limited simplification
  --! BY synthesis of TRUE
  return (s.top=0);
end empty;
```

A.4.4 Proof of procedure push

In procedure push we have to verify the effect of the push, if it occurs, but we also have to verify that exceptional termination occurs when, and only when, the stack is full.

In the verification condition for the subprogram, we appeal to theorem *push* of trait *StackImpl* to show the effect of the push, and to theorem *size* to compute the size of the stack before and after the push. Theorems *size* and *limit* are also used to show that the test in the **if** statement is correct. The invocation of *limited-simplify* causes the rewrite rules to be applied. Note that *abs_stack* is not appealed to directly, but only through *push* and *size*. Those lemmas shorten the proof.

We use in-line simplification to show that `stack_full` is raised correctly. The lemma that justifies the code here is *size* in trait *StackImpl*. Information needed to reduce the precondition to **true** is not available at this point in the program, so the unproved sequents are hidden (<>).

```
procedure push(n :  in elem; s :  in out stack)
  --| where * * *
  --|    out (s=push(n, in s));
  --|    raise stack_full <=> in (size(s)=stack_limit);
  --| end where;
  --!    rewrite rule: inv_stack in trait StackImpl;
  --! VC Status: proved
  --! BY instantiation of push in trait StackImpl
  --! rewriting right to left
  --! BY instantiation of size in trait Stacks
  --! rewriting left to right
  --! BY limited simplification
  --! BY instantiation of size in trait StackImpl establishing
  --!    BY simplification
  --!    BY synthesis of TRUE
  --! THEN
  --! rewriting left to right
```

```
  --! BY simplification
  --! BY synthesis of TRUE
is
begin
  if (s.top=stack_limit) then
    raise stack_full;
  end if;
  s.top:=(s.top+1);
  s.contents(s.top):=n;
end push;
```

A.4.5 Proof of procedure pop

The theorems *top* and *pop* of trait *StackImpl* justify the assignment statements. We invoke them close to the statements that they justify. We could just as well put all of the proof of this short subprogram in the verification condition. The theorem *empty* of trait *StackImpl* is used to assure that the exception is raised as specified in the exact propagation constraint. Theorem *size* is needed to show that after popping, the stack is still within its upper limit.

```
procedure pop(result : out elem; s : in out stack)
  --| where * * *
  --|   out (s=pop(in s));
  --|   out result=top(in s);
  --|   raise stack_empty <=> in (s=empty());
  --| end where;
  --! rewrite rule: inv_stack in trait StackImpl;
  --! rewrite rule: empty in trait StackImpl;
  --! VC Status: proved
  --! BY limited simplification
  --! BY instantiation of size in trait StackImpl establishing
  --!   BY simplification
  --!   BY synthesis of TRUE
  --! THEN
  --! rewriting left to right
  --! BY instantiation of size in trait StackImpl establishing
  --!   BY limited simplification, simplification
  --!   BY synthesis of TRUE
  --! THEN
  --! rewriting left to right
  --! BY simplification
  --! BY synthesis of TRUE
is
begin
  if (s.top=0) then
```

```
      raise stack_empty;
    end if;
      result:=s.contents(s.top);
      s.top:=(s.top-1);
    --! BY instantiation of pop in trait StackImpl establishing
    --!    <>
    --! THEN
    --! rewriting left to right
    --! BY instantiation of top in trait StackImpl establishing
    --!    <>
    --! THEN
    --! rewriting left to right
    --!    <>
  end pop;
  end stack;
```

Appendix B

Subset of Ada supported

B.1 Introduction

This appendix informally describes the subset of Ada supported by Penelope at the time of publication. The organization of this appendix follows that of [1]. Section numbers correspond to chapter and section numbers in [1]. Many of the features not yet supported by the software are supported by the theory [19].

Penelope uses an abstract syntax for Ada. Thus the syntax may not agree with that given in [1].

Penelope assumes correct Ada code, but does not guarantee it. Penelope performs some static semantic checking, but the checking is not complete. When static checks fail, Penelope abandons the verification effort and replaces the current precondition with **undefined** (see Section 5.11).

In order to assure that programs are free of *incorrect order dependence*, Penelope requires that certain values be *independent*. That is, if $reads(E)$ are the program objects potentially read during evaluation of E and $writes(E)$ are the program objects potentially written during evaluation of E, then E_1 and E_2 are independent if and only if

$$\begin{aligned} reads(E_1) \cap writes(E_2) &= \emptyset \wedge \\ writes(E_1) \cap reads(E_2) &= \emptyset \wedge \\ writes(E_1) \cap writes(E_2) &= \emptyset \end{aligned}$$

Independence is a sufficient, although not a necessary, condition for avoiding incorrect order dependence errors. In independence requirements, entire objects are considered: record `r` and and array `a` are considered to be single objects. Thus if one expression causes `a(i)` to be modified and another reads `a(j)`, the two expressions are not independent, even if `i` and `j` are known to be distinct. The current version of Penelope may not issue warnings when independence requirements are violated.

B.2 Lexical elements

Penelope supports the character set of Ada. In addition to the compound delimiters of Ada, Penelope supports the compound delimiters described in Section 3.2 of this manual.

Identifiers, numeric literals, and decimal literals are as in Ada, except that underlines may not occur within integers or real literals.

Comments are not supported at arbitrary textual positions in a program. Comments may appear wherever a declaration or a statement may appear.

Pragma elaborate is the only pragma supported.

The reserved words of Ada are reserved in Penelope. In addition, Penelope reserves the words in Section 3.5, which may not be used as identifiers in programs verified by Penelope.

B.3 Declarations and types

B.3.1 Declarations

Penelope supports the basic declarations of Ada. A major restriction is that subtypes are not supported.

Comments are acceptable everywhere a declaration is acceptable.

$$\langle basic\_declarative\_item \rangle ::= \quad \langle comment \rangle$$

B.3.2 Objects and named numbers

Object declarations are supported.

$$\begin{array}{l} \langle basic\_declarative\_item \rangle ::= \\ \quad \langle idlist \rangle : \langle typemark \rangle \; [[\langle initialization \rangle]] \; ; \\ \langle idlist \rangle ::= \quad [[\langle identifier \rangle]]^{+}_{,} \\ \langle initialization \rangle ::= \quad := \langle expression \rangle \end{array}$$

Constant declarations are not supported. Note that Penelope does not yet check that variables are initialized before use.

B.3.3 Types and subtypes

Penelope supports the types `boolean`, `integer`, and `float`, as well as array and record types. There are no anonymous types—all types must be declared. Subtypes are not supported. Two attributes of types (T'FIRST and T'LAST) are supported. Discriminant parts of types are not supported.

$\langle$*basic_declarative_item*$\rangle$ ::=
  type $\langle$*identifier*$\rangle$ **is** $\langle$*type_definition*$\rangle$;

B.3.4 Derived types

Derived types are not supported.

B.3.5 Scalar types

The types `boolean`, `integer`, and `float` are supported. User-defined enumeration types are supported. Relational operators are defined on numeric and enumeration types, but not for type `boolean`. Character types are not supported.

B.3.6 Array types

Penelope supports constrained arrays of multiple dimensions. Unconstrained arrays and strings are not supported. Index types must be types accepted by Penelope, hence ranges are not supported.

$\langle$*type_definition*$\rangle$ ::=
  array ($[[\langle \textit{typemark} \rangle]]^{+}_{,}$) **of** $\langle$*typemark*$\rangle$

B.3.7 Record types

Record types are supported, but record variants and discriminant parts are not. Default values for record fields are not supported.

$\langle$*type_definition*$\rangle$ ::=
  record
    $[[\langle \textit{component\_declaration} \rangle]]^{*}$
  end record
$\langle$*component_declaration*$\rangle$ ::=
  $\langle$*identifier*$\rangle$: $\langle$*typemark*$\rangle$;

B.3.8 Access types

Access types are not supported.

B.3.9 Declarative parts

$\langle \textit{declarative\_part} \rangle$ ::=
  $[[\langle \textit{basic\_declarative\_item} \rangle]]^*$
  $[[\langle \textit{later\_declarative\_item} \rangle]]^*$

$\langle \textit{basic\_declarative\_item} \rangle$ ::=
  $\langle \textit{subprogram\_declaration} \rangle$
  | $\langle \textit{package\_declaration} \rangle$

$\langle \textit{later\_declarative\_item} \rangle$ ::=
  $\langle \textit{subprogram\_declaration} \rangle$
  | $\langle \textit{package\_declaration} \rangle$
  | $\langle \textit{subprogram\_body} \rangle$
  | $\langle \textit{package\_body} \rangle$

Elaboration of subprogram declarations and bodies is not verified.

B.4 Names and expressions

B.4.1 Names

Penelope supports simple names, indexed components, and selected components, including function call and expanded names. Function calls may occur in the prefix of an indexed component or selected component. Slices are not supported.

$\langle \textit{name} \rangle$ ::=
  $\langle \textit{identifier} \rangle$
  | $\langle \textit{name} \rangle$ $([[\langle \textit{explist} \rangle]]^+_,)$
  | $\langle \textit{name} \rangle.\langle \textit{identifier} \rangle$

The independence requirements (see Section B.1) for the name $\mathtt{A}(\mathtt{B}_1, \ldots, \mathtt{B}_n)$ are

- For an array, A must be independent of all $\mathtt{B}_i$.
- All $\mathtt{B}_i$ must be pairwise independent.

In addition, all function arguments must be pairwise distinct (see Section B.6.4).

B.4.2 Literals

Numeric literals are supported. Character, string, and enumeration literals (other than **true** and **false**) are not supported, nor is the literal **null**.

B.4.3 Aggregates

Aggregates are not supported.

B.4.4 Expressions

Expressions are supported. Short circuit control forms are not supported.

B.4.5 Operators and expression evaluation

All Ada unary and binary operators are supported, except for the short circuit control forms and catenation.

⟨*expression*⟩ ::=
  ⟨*integer_literal*⟩
| ⟨*real_literal*⟩
| ⟨*name*⟩
| ⟨*unary_operator*⟩ ⟨*expression*⟩
| ⟨*expression*⟩ ⟨*binary_operator*⟩ ⟨*expression*⟩
⟨*unary_operator*⟩ ::=
  + | − | **abs** | **not**
⟨*binary_operator*⟩ ::=
  and | **or** | **xor**
  | = | /= | < | <= | > | >=
  | + | − | & | * | / | **mod** | **rem** | **

Operands of binary operators must be independent, in the sense that objects written by evaluation of the left operand must not include any objects read or written by evaluation of the right operand and vice versa.

B.4.6 Type conversions

Type conversions are not supported.

B.4.7 Qualified expressions

Qualified expressions are not supported.

B.4.8 Allocators

Allocators are not supported.

B.4.9 Static expressions and static subtypes

This topic is not relevant.

B.4.10 Universal expressions

This topic is not relevant.

B.5 Statements

Penelope supports most popular control structures. Labels are not supported, since **goto** statements are not supported.

B.5.1 Null, pseudo-statements, and sequences of statements

$\langle statement\_sequence \rangle$ ::=
  $[[\langle statement \rangle]]^{+}$
$\langle statement \rangle$ ::=
  null;
  | $\langle comment \rangle$
  | $\langle embedded\_assertion \rangle$;
  | $\langle cut\_point\_assertion \rangle$;

Embedded and cut point assertions can appear wherever a statement can appear, as can comments.

B.5.2 Assignment statement

⟨*statement*⟩ ::=
  ⟨*name*⟩ := ⟨*expression*⟩;

Penelope requires that the name and the expression be independent; that is, the evaluation of one does not write any objects that may be read or written in the evaluation of the other. Thus if `x` and `i` are variables, `a` is an array and `f` is a function,

```
x:=x+1;
```

is legal. If `f` has a side effect on `i`, then

```
a(i)=f(x);
```

is not legal, because the expression writes `i`, while evaluation of the name on the left must read it.

B.5.3 If statements

⟨*statement*⟩ ::=
  if ⟨*expression*⟩ **then**
    ⟨*statement_sequence*⟩
  [[**elsif** ⟨*expression*⟩ **then**
    ⟨*statement_sequence*⟩]]*
  [[**else**
    ⟨*statement_sequence*⟩]]
  end if;

B.5.4 Case statements

Case statements are supported.

⟨*statement*⟩ ::=
  if ⟨*expression*⟩ **is**
    [[⟨*case_statement_alternative*⟩]]$^+$
  end case;
⟨*case_statement_alternative*⟩ ::=
  when [[⟨*choice*⟩]]$^+_|$ **=>**
    ⟨*statement_sequence*⟩

B.5.5 Loop statements

⟨*statement*⟩ ::=
  ⟨*verification_condition*⟩
  [[⟨*loop_name*⟩]] [[⟨*iter_scheme*⟩]] **loop**
    --| invariant ⟨*term*⟩;
    ⟨*statement_sequence*⟩
  end loop [[⟨*identifier*⟩]];
⟨*loop_name*⟩ ::= ⟨*identifier*⟩ :
⟨*iter_scheme*⟩ ::=
  while ⟨*expression*⟩
  | **for** ⟨*identifier*⟩ **in** [[**reverse**]] ⟨*forloop_range*⟩
⟨*forloop_range*⟩ ::=
  ⟨*expression*⟩ .. ⟨*expression*⟩

If a loop name is present, Penelope repeats the name at the end of the loop.

B.5.6 Block statements

⟨*statement*⟩ ::=
  [[⟨*block_name*⟩]] [[⟨*block_declare*⟩]]
  begin
    ⟨*statement_sequence*⟩
    [[⟨*exception_handling*⟩]]
  end ⟨*identifier*⟩
⟨*block_name*⟩ ::= ⟨*identifier*⟩ :
⟨*block_declare*⟩ ::=
  declare
    ⟨*declarative_part*⟩

B.5.7 Exit statements

$\langle statement \rangle$ ::=
  exit [[$\langle exit\_name \rangle$]] [[$\langle exit\_when \rangle$]];
$\langle exit\_name \rangle$ ::= $\langle identifier \rangle$
$\langle exit\_when \rangle$ ::= **when** $\langle expression \rangle$

B.5.8 Return statements

$\langle statement \rangle$ ::=
  return [[$\langle expression \rangle$]];

B.5.9 Goto statements

Goto statements are not supported.

B.6 Subprograms

Subprograms, including mutually recursive subprograms, are supported.

B.6.1 Subprogram declarations

$\langle subprogram\_declaration \rangle$ ::=
  $\langle subprogram\_spec \rangle$;
  $\langle subprogram\ annotation \rangle$
$\langle subprogram\_spec \rangle$ ::=
  procedure $\langle identifier \rangle$ [[$\langle formal\_part \rangle$]]
  | **function** $\langle designator \rangle$ [[$\langle formal\_part \rangle$]] **return** $\langle type\_mark \rangle$
$\langle formal\_part \rangle$ ::=
  ([[$\langle idlist \rangle$: $\langle mode \rangle$ $\langle type\_mark \rangle$]]$^{+}_{;}$)
$\langle mode \rangle$ ::=
  in | **in out** | **out**

Operator symbols as function designators are supported. Default expressions for parameters are not supported.

The parameters of a subprogram include its formal parameters and also its *global parameters*, objects that it potentially reads or writes during execution. Global parameters must be declared in the subprogram annotation (see Section 6.1).

B.6.2 Formal parameter modes

The modes **in**, **out**, and **in out** are supported.

B.6.3 Subprogram bodies

⟨*subprogram_body*⟩ ::=
  ⟨*subprogram_spec*⟩
    ⟨*subprogram annotation*⟩
  is
    ⟨*declarative_part*⟩
  begin
    ⟨*statement_sequence*⟩
    [[⟨*exception_handling*⟩]]
  end ⟨*identifier*⟩;

Penelope does not (yet) support elaboration of subprogram declarations and bodies.

If a subprogram has a declaration, Penelope automatically copies its subprogram annotation to the body. You can replace this default subprogram annotation. In this case, a verification condition assures that the annotations of the declaration and body are consistent.

B.6.4 Subprogram calls

⟨*statement*⟩ ::=
  ⟨*name*⟩ [[⟨*actual_parameter_part*⟩]];
⟨*actual_parameter_part*⟩ ::=
  ([[expression]]$^{+}_{,}$)

Named parameter associations are not supported. Type conversions are not supported. Default parameters are not supported.

All global and formal arguments must be distinct. Note that in this context, "argument" refers to a declared object, not a component or selected component. Thus if `swap` is a procedure with two **in out** parameters and `a` is an array, then `swap(a(i),a(j))` is not allowed, because `a` is the object in both parameters.

B.6.5 Function subprograms

Function calls have the same syntax as simple names (no arguments) or indexed components.

B.6.6 Overloading of subprograms

Penelope supports overloading of subprograms.

B.6.7 Overloading of operators

Penelope supports overloading of operators.

B.7 Packages

Penelope supports packages and private types.

B.7.1 Package structure

Packages may occur as declarations or as compilation units.

$\langle$*package_spec*$\rangle$ ::=
  package $\langle$*identifier*$\rangle$ **is**
    [[$\langle$*basic_declarative_item*$\rangle$]]*
  [[$\langle$*private_part*$\rangle$]]
  end $\langle$*identifier*$\rangle$;

$\langle$*package_spec*$\rangle$ ::=
  package body $\langle$*identifier*$\rangle$ **is**
    $\langle$*declarative_part*$\rangle$
  [[$\langle$*statement_part*$\rangle$]]
  end $\langle$*identifier*$\rangle$;
$\langle$*statement_part*$\rangle$ ::=
  begin
    $\langle$*statement_sequence*$\rangle$
    [[$\langle$*exception_handling*$\rangle$]]

B.7.2 Package specifications and declarations

Penelope verifies the elaboration of package declarations.

B.7.3 Package bodies

Penelope verifies the elaboration of package bodies. Variables may be declared in the body of a package, but Penelope does not yet support annotations that would make it possible to

observe the value of such a variable from outside the package. Specifically, in subprogram annotations in the package declaration, it is not yet possible to refer to the values of such internal (not visible) variables.

B.7.4 Private type and deferred constant declarations

Deferred constants and limited private types are not supported.

⟨*type_definition*⟩ ::=
  private;
    --| based on ⟨*sortmark*⟩

The implementation of a private type in the private part of a package must include an abstraction function and representation invariant (see Section 6.4.1). To obtain a template for such a declaration, click on `private-type-implementation` on the help-pane menu.

B.8 Visibility rules

Penelope supports Ada's rules concerning scope of declaration and visibility.

B.8.1 Declarative region

Declarative regions are supported.

B.8.2 Scope of declarations

Penelope supports Ada's scope rules.

B.8.3 Visibility

Penelope supports direct visibility. Visibility by selection is partially supported: objects within packages are generally visible by selection.

B.8.4 Use clauses

Use clauses are not supported.

B.8.5 Renaming declarations

Renaming declarations are not supported.

B.8.6 The package `standard`

The following are predefined in Penelope:

- types `boolean`, `integer`, `float`
- predefined relational operators on types `integer` and `float`
- the exception `program_error`

The exception `program_error` is supported in that verification will fail when `program_error` may be raised by a program.

B.8.7 The context of overload resolution

Penelope performs overload resolution and flags constructs in which a subprogram or operator is ambiguous.

B.9 Tasks

Tasks are not supported.

B.10 Program structure and compilation issues

In Penelope, traits are treated as compilation units. The method of specifying a main program is described in Section 6.5.5.

B.10.1 Compilation units—library units

$\langle compilation \rangle$ ::=
  [[$\langle compilation\_unit \rangle$]]*
$\langle compilation\_unit \rangle$ ::=
  $\langle trait \rangle$
  | [[$\langle context\_clause \rangle$]] [[$\langle pragma\_elaborate \rangle$]]
   $\langle compilation\_unit\_proper \rangle$
$\langle compilation\_unit\_proper \rangle$ ::=
  $\langle subprogram\_decl \rangle$
 | $\langle subprogram\_body \rangle$
 | $\langle package\_decl \rangle$
 | $\langle package\_body \rangle$

B.10.2 Subunits of compilation units

Subunits are not supported.

B.10.3 Order of compilation

The order of verification must be consistent with the partial ordering defined for compilation, except that you can update the library at any time (even with an incomplete verification). The same considerations apply for reverification.

In the current version of Penelope you are responsible for verifying compilation units in a correct order. See item 5 of Section 2.6 for a discussion of the order of verification.

B.10.4 The program library

Penelope library support is discussed in Section 2.5.

B.10.5 Elaboration of library units

Ada defines constraints on the order of elaborating library units prior to the execution of a main program. In Penelope, a conservative check is made to ensure that every legal order of elaborating library units gives the same result (i.e., that there is no incorrect order dependence in the elaboration). Specifically, if the partial ordering for elaboration does not define the order of elaboration of library units `A` and `B`, then `A` and `B` must be independent: That is, let $reads(E)$ be the program objects potentially read during elaboration of E and $writes(E)$ be the program objects potentially written during elaboration of E. Then A and B are independent if and only if

$$reads(A) \cap writes(B) \;=\; \emptyset \wedge$$

$$\begin{aligned} writes(A) \cap reads(B) &= \emptyset \wedge \\ writes(A) \cap writes(B) &= \emptyset \end{aligned}$$

Penelope supports pragma `elaborate` to ensure prior elaboration of library unit bodies.

⟨*pragma_elaborate*⟩ ::=
  pragma elaborate ⟨*idlist*⟩;

B.10.6 Program optimization

Penelope assumes that optimization that changes the effect of execution of the program does not occur.

B.11 Exceptions

B.11.1 Exception declarations

⟨*basic_declarative_item*⟩ ::=
  ⟨*idlist*⟩: **exception**;

User-defined exceptions are supported. The predefined exception `program_error` is supported, in the sense that programs that raise `program_error` cannot be verified. Penelope assumes that predefined exceptions `constraint_error`, `numeric_error`, `storage_error`, and `tasking_error` are not raised, and its verification conditions do not cover these exceptions.

Syntactically, predefined exceptions can be named in exception choices, but currently Penelope does not raise them.

B.11.2 Exception handlers

⟨*exception_handling*⟩ ::=
  when [[⟨*exception_choice*⟩]]$^+$ =>
    ⟨*statement_sequence*⟩
⟨*exception_choice*⟩ ::= ⟨*name*⟩ | **others**

B.11.3 Raise statements

$\langle statement \rangle$ ::=
raise [[$\langle name \rangle$]];

A template is available for a raise statement without an exception name; click `raise-again` on the help-pane menu.

B.11.4 Exception handling

Exception handling is supported during the execution of a sequence of statements and the elaboration of declarations. Penelope assures that no exception is raised during the elaboration of library units.

B.11.5 Exceptions raised during task communication

Not applicable.

B.11.6 Exceptions and optimization

Penelope assumes that optimization that changes the effect of execution of the program does not occur.

B.11.7 Suppressing checks

A compiler that operates in conjunction with a verification system may omit checks for exceptions that are provably not raised. Suppressing arbitrary checks is illegal.

B.12 Generic units

Generic subprograms and packages are supported. Formal objects of mode **in** and formal private types and array types are supported. Formal objects of mode other than **in** are not supported. Generic formal subprograms are not supported, nor are formal floating point, fixed point or limited private types.

⟨*generic_specification*⟩ ::=
  ⟨*generic_formal_part*⟩ ⟨*package_specification*⟩
| ⟨*generic_formal_part*⟩ ⟨*subprogram_specification*⟩
⟨*generic_formal_part*⟩ ::=
  generic
    [[⟨*formal_trait*⟩]]
    ⟨*generic_parameter_decls*⟩
⟨*formal_trait*⟩ ::=
  --| formal trait
    ⟨*trait body*⟩
  --| end formal trait
⟨*generic_parameter_decls*⟩ ::=
  [[⟨*generic_parameter_decl*⟩]]*
⟨*generic_parameter_decl*⟩ ::=
  ⟨*idlist*⟩ [[**in**]] ⟨*typemark*⟩;
| ⟨*private_type_declaration*⟩
| **type** ⟨*identifier*⟩ **is** ⟨*generic_type_definition*⟩;
| ⟨*generic_assertion*⟩;
⟨*generic_type_definition*⟩ ::= **(<>)** | **range <>**
⟨*generic_assertion*⟩ ::=
  --| ⟨*identifier*⟩ : ⟨*term*⟩ ;

Penelope supports only positional parameter associations for generic instantiation. Named associations are not supported.

⟨*generic_instantiation*⟩ ::=
  ⟨*instantiation_kind*⟩ ⟨*identifier*⟩ **is new** ⟨*name*⟩ [[⟨*generic_actual_part*⟩]];
  [[⟨*fitting_morphism*⟩]]
⟨*instantiation_kind*⟩ ::=
  function
| **procedure**
| **package**
⟨*generic_actual_part*⟩ ::=
  ($[[\text{generic\_actual\_parameter}]]^{+}_{,}$)
⟨*generic_actual_parameter*⟩ ::=
  ⟨*expression*⟩
⟨*fitting_morphism*⟩ ::=
  --| ⟨*renaming_list*⟩

B.13 Representation clauses and implementation-dependent features

Not applicable.

B.14 Input-output

Penelope does not support Ada input-output constructs.

Appendix C

Summary of proof rules

Menu access via	**Meaning**
`add-as-hypothesis`	$\dfrac{\Gamma, P^{\bar{x}}_{\bar{v}} \Rightarrow Q, \text{P a theorem}}{\Gamma \Rightarrow Q}$
`add-as-rewrite-rule`	Adds instantiated theorem of form $l = r$ as rewrite rule. See page 83.
`add-as-reversed -rewrite-rule`	Adds instantiated theorem of form $l = r$ as rewrite rule $r = l$. See page 83.
`affirmation-synthesis`	$\dfrac{\Gamma \Rightarrow Q}{\Gamma \Rightarrow Q = \textbf{true}}$
`analyze-hypothesis`	Apply a rule based on the syntax of one hypothesis.
`and-anal`	$\dfrac{\Gamma, P_1, \ldots, P_n \Rightarrow Q}{\Gamma, P_1 \wedge \ldots \wedge P_n \Rightarrow Q}$
`and-syn`	$\dfrac{\Gamma \Rightarrow Q_1, \ldots, \Gamma \Rightarrow Q_n}{\Gamma \Rightarrow Q_1 \wedge \ldots \wedge Q_n}$

Summary of proof rules, continued

`approximate-simplify`	Apply Nelson-Oppen simplifier to real arithmetic. See page 77.
`approximate-simplify -conclusion`	Apply Nelson-Oppen simplifier for real arithmetic to conclusion only. See page 77.
`array-simplification`	Expand array references of the form $a[i{=}{>}v][j]$ to **if** $i{=}j$ **then** v **else** $a[j]$. See page 77.
`arithmetic`	$\overline{\Gamma \Rightarrow Q}$,where Q is an axiom of arithmetic
`assume`	$\dfrac{\text{Because I said so}}{\Gamma \Rightarrow Q}$
`case`	$\dfrac{\Gamma, P \Rightarrow Q \qquad \Gamma, \neg P \Rightarrow Q}{\Gamma \Rightarrow Q}$
`claim`	$\dfrac{\Gamma \Rightarrow P \qquad \Gamma, P \Rightarrow Q}{\Gamma \Rightarrow Q}$
`conflicting-hypotheses`	$\overline{\Gamma, P, \neg P \Rightarrow Q}$
`contradiction`	$\dfrac{\Gamma, \neg Q \Rightarrow \mathbf{false}}{\Gamma \Rightarrow Q}$
`contradiction (hypothesis)`	$\dfrac{\Gamma, \neg Q \quad \Rightarrow \neg P}{\Gamma, P \quad \Rightarrow Q}$
`denial-synthesis`	$\dfrac{\Gamma \Rightarrow \neg Q}{\Gamma \Rightarrow Q = \mathbf{false}}$
`direct-subst`	$\dfrac{\Gamma, \Rightarrow Q_P^{\mathbf{true}}}{\Gamma, P \Rightarrow Q}$
`distribution`	Distribute multiplication over addition.

Summary of proof rules, continued

`equals-analysis`	$\dfrac{\Gamma, l = r, \Gamma_l^r \quad \Rightarrow Q_l^r}{\Gamma, l = r \quad \Rightarrow Q}$
`equals-synthesis`	$\dfrac{\Gamma, Q_1 \Rightarrow Q_2 \quad \Gamma, Q_2 \Rightarrow Q_1}{\Gamma \Rightarrow Q_1 = Q_2, \text{where } Q_1,\ Q_2 \text{ boolean.}}$
`exists-anal`	$\dfrac{\Gamma, \textbf{exists } x :: P, P_x^y \Rightarrow Q, \quad \text{y not free in } Q \text{ or } \Gamma}{\Gamma, \textbf{exists } x :: P \Rightarrow Q}$
`exists-syn`	$\dfrac{\Gamma \Rightarrow Q_x^y}{\Gamma \Rightarrow \textbf{exists } x :: Q}$
`explicit-roundoff`	For real arithmetic, introduce explicit expressions involving rounding.
`extensionality`	See Section 8.10.
`false-anal`	$\overline{\Gamma, \textbf{false} \Rightarrow Q}$
`forall-anal`	$\dfrac{\Gamma, \textbf{forall } x :: P, P_x^y \quad \Rightarrow Q}{\Gamma, \textbf{forall } x :: P \quad \Rightarrow Q}$
`forall-syn`	$\dfrac{\Gamma \Rightarrow Q_x^y, \text{where } y \text{ not free in } \Gamma \text{ or (if } y \neq x) \text{ Q}}{\Gamma \Rightarrow \textbf{forall } x :: Q}$
`forall/implies-syn`	See page 86.
`hide-sequent`	Do not display sequent. Do not continue proof here.
`hypothesis`	$\overline{\Gamma, Q \Rightarrow Q}$
`if-branch-selection`	See page 87.

Summary of proof rules, continued

`if-else-anal`	$\dfrac{\Gamma \Rightarrow \neg P \qquad \Gamma, \neg P, S \Rightarrow Q}{\Gamma, \textbf{if } P \textbf{ then } R \textbf{ else } S \Rightarrow Q}$
`if-pair`	$\dfrac{\Gamma, P, A \Rightarrow Q \qquad \Gamma, \neg P, B \Rightarrow R}{\Gamma, \textbf{if } P \textbf{ then } A \textbf{ else } B \Rightarrow \textbf{if } P \textbf{ then } Q \textbf{ else } R}$
`if-syn`	$\dfrac{\Gamma, P \Rightarrow R \qquad \Gamma, \neg P \Rightarrow S}{\Gamma \Rightarrow \textbf{if } P \textbf{ then } R \textbf{ else } S}$
`if-then-anal`	$\dfrac{\Gamma \Rightarrow P \qquad \Gamma, P, R \Rightarrow Q}{\Gamma, \textbf{if } P \textbf{ then } R \textbf{ else } S \Rightarrow Q}$
`imp-anal`	$\dfrac{\Gamma \Rightarrow P \qquad \Gamma, Q \Rightarrow R}{\Gamma, P \rightarrow Q \Rightarrow R}$
`induction`	See Section 8.9.
`limited-simplify`	See Section 8.4 and page 77.
`not-analysis`	$\dfrac{\Gamma \Rightarrow P}{\Gamma, \neg P \Rightarrow Q}$
`not-equals-analysis`	$\dfrac{\Gamma, x \neq y, \neg(x = y) \Rightarrow Q}{\Gamma, x \neq y \Rightarrow Q}$
`not-equals-synthesis`	$\dfrac{\Gamma \Rightarrow \neg(x = y)}{\Gamma \Rightarrow x \neq y}$
`not-syn`	$\dfrac{\Gamma, Q \Rightarrow \textbf{false}}{\Gamma \Rightarrow \neg Q}$
`or-anal`	$\dfrac{\Gamma, P_1 \Rightarrow Q \qquad \Gamma, P_2 \Rightarrow Q}{\Gamma, P_1 \vee P_2 \Rightarrow Q}$
`or-syn-1`	$\dfrac{\Gamma, \neg Q_2 \Rightarrow Q_1}{\Gamma \Rightarrow Q_1 \vee Q_2}$

Summary of proof rules, continued

`or-syn-r`

$$\frac{\Gamma, \neg Q_1 \Rightarrow Q_2}{\Gamma \Rightarrow Q_1 \vee Q_2}$$

`prenex-simplify`

Not implemented.

`rewrite-left-to-right`

$$\frac{\Gamma \Rightarrow c^{\vec{x}}_{\vec{v}}; \quad \Gamma, \Gamma^{r'}_{l'} \Rightarrow Q^{r'}_{l'}}{\Gamma \Rightarrow Q},$$

where $c \rightarrow l = r$ is a theorem with free variables $\vec{v}$; $r' = r^{\vec{x}}_{\vec{v}}$, and $l' = l^{\vec{x}}_{\vec{v}}$. See page 81.

`rewrite-right-to-left`

$$\frac{\Gamma \Rightarrow c^{\vec{x}}_{\vec{v}}; \quad \Gamma, \Gamma^{l'}_{r'} \Rightarrow Q^{l'}_{r'}}{\Gamma \Rightarrow Q},$$

where $c \rightarrow l = r$ is a theorem with free variables $\vec{v}$, $r' = r^{\vec{x}}_{\vec{v}}$, and $l' = l^{\vec{x}}_{\vec{v}}$. See page 82.

`rewrite-to-true`

$$\frac{\Gamma \Rightarrow Q^{\mathbf{true}}_{P^{\vec{x}}_{\vec{v}}}, P \text{ a theorem}}{\Gamma \Rightarrow Q}$$

`self-identity`

$$\frac{}{\Gamma \Rightarrow x = x}$$

`SDVS-simplify`

Apply Nelson-Oppen simplifier. See page 76.

`SDVS-simplify -conclusion`

Apply Nelson-Oppen simplifier to conclusion only. See page 76.

`simplify`

Apply one of the simplification methods. See Section 8.3.

Summary of proof rules, continued

`synthesize-conclusion`	Apply a rule based on the syntax of the sequent's conclusion.
`thinning`	$\dfrac{\Gamma \Rightarrow Q}{\Gamma, P \Rightarrow Q}$
`true-syn`	$\overline{\Gamma \Rightarrow \textbf{true}}$
`xor-anal`	$\dfrac{\Gamma, (P_1 \wedge \neg P_2) \vee (\neg P_1 \wedge P_2) \Rightarrow Q}{\Gamma, P_1 \oplus P_2 \Rightarrow Q}$
`xor-syn`	$\dfrac{\Gamma \Rightarrow (Q_1 \wedge \neg Q_2) \vee (\neg Q_1 \wedge Q_2)}{\Gamma \Rightarrow Q_1 \oplus Q_2}$

Bibliography

[1] ANSI. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A.

[2] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[3] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science XIX*, pages 19–32. American Mathematical Society, 1967.

[4] Michael J. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, July 1985.

[5] GrammaTech. *The Synthesizer Generator Reference Manual*, release 4.1 edition, 1993.

[6] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[7] David Guaspari. Formal definition of satisfaction. Technical report, ORA, November 1988.

[8] David Guaspari. Domains for Ada types. Technical report, ORA, September 1989.

[9] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16:1058–1075, September 1990.

[10] J. V. Guttag, J. J. Horning, and Andres Modet. Report on the larch shared language. Technical report, DEC/SRC, April 1990.

[11] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[12] C. Douglas Harper. The logical foundation of Penelope. Technical report, ORA, September 1989.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[14] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[15] M.E. Lesk. Lex—a lexical analyzer generator. Technical Report Comp. Sci. Tech Report 39, Bell Laboratories, Murray Hill, NJ, 1975.

[16] Carla Marceau and C. Douglas Harper. An interactive approach to Ada verification. In *Proceedings of the 12th National Computer Security Conference*, pages 28–51, Baltimore, MD, October 1989.

[17] ORA. Larch/Ada rationale. Technical report, ORA, September 1989.

[18] Wolfgang Polak. Predicate Transformer Semantics for Ada, release 1.5, September 1992.

[19] Wolfgang Polak. Predicate Transformer Semantics for Ada, release 1.6, September 1992.

Index